VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

AKCELERACE KOMPRESNÍHO ALGORITMU LZ4 V FPGA

ACCELERATION OF LZ4 COMPRESSION ALGORITHM IN FPGA

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. DOMINIK MARTON**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. JIŘÍ MATOUŠEK**
SUPERVISOR

BRNO 2017

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2016/2017

Zadání diplomové práce

Řešitel: **Marton Dominik, Bc.**
Obor: Počítačové sítě a komunikace
Téma: **Akcelerace kompresního algoritmu LZ4 v FPGA**
Acceleration of LZ4 Compression Algorithm in FPGA
Kategorie: Návrh číslicových systémů

Pokyny:

1. Seznamte se s principy bezztrátových kompresních algoritmů. Zaměřte se především na slovníkové algoritmy a jejich variantu LZ4.
2. Proveďte srovnání těchto algoritmů na základě dostupných parametrů (kompresní poměr, rychlost komprese/dekomprese, aj.).
3. Navrhněte moduly implementující LZ4 kompresi a dekompresi v FPGA.
4. Navržené moduly implementujte s využitím jazyka Catapult C a ověřte jejich funkčnost oproti softwarové implementaci.
5. Experimentálně ověřte parametry implementovaných modulů.
6. Diskutujte zjištěné parametry modulů a proveďte jejich srovnání se softwarovou implementací LZ4 komprese a dekomprese.

Literatura:

- Dle pokynů vedoucího práce.

Při obhajobě semestrální části projektu je požadováno:

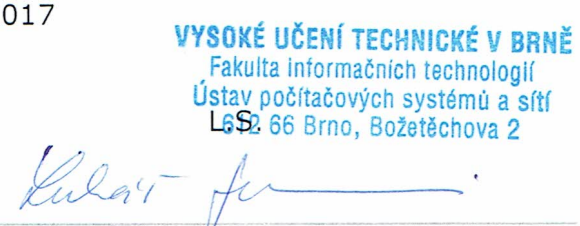
- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Matoušek Jiří, Ing.**, UPSY FIT VUT
Konzultant: Dvořák Milan, Ing., UPSY FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 24. května 2017



prof. Ing. Lukáš Sekanina, Ph.D.
vedoucí ústavu

Abstrakt

Tato práce popisuje implementaci kompresního algoritmu LZ4 v syntetizovatelném jazyce z rodiny C/C++, pomocí kterého je možné získat VHDL kód pro FPGA čipy na síťových kartách. Podle specifikace algoritmu je implementovaná softwarová verze kompresoru a dekompresoru, která je poté transformována do syntetizovatelného jazyka, ze kterého je vygenerován plně funkční VHDL kód obou komponent. Jednotlivé implementace jsou poté porovnány na základě doby běhu a kompresního poměru. Práce demonstruje význam a sílu high-level syntézy a vysokoúrovňového přístupu z klasických programovacích jazyků při návrhu a implementaci aplikací v hardwaru.

Abstract

This project describes the implementation of an LZ4 compression algorithm in a C/C++-like language, that can be used to generate VHDL programs for FPGA integrated circuits embedded in accelerated network interface controllers (NICs). Based on the algorithm specification, software versions of LZ4 compressor and decompressor are implemented, which are then transformed into a synthesizable language, that is then used to generate fully functional VHDL code for both components. Execution time and compression ratio of all implementations are then compared. The project also serves as a demonstration of usability and influence of high-level synthesis and high-level approach to design and implementation of hardware applications known from common programming languages.

Klíčová slova

rychlá bezeztrátová komprese, LZ4, slovníkové kompresní algoritmy, FPGA, Catapult, high-level syntéza

Keywords

fast lossless compression, LZ4, dictionary-based compression algorithms, FPGA, Catapult, high-level synthesis

Citace

MARTON, Dominik. *Akcelerace kompresního algoritmu LZ4 v FPGA*. Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jiří Matoušek

Akcelerace kompresního algoritmu LZ4 v FPGA

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Jiřího Matouška a uvedl všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Dominik Marton
23. května 2017

Poděkování

Děkuji vedoucímu této diplomové projektu, panu Ing. Jiřímu Matouškovi, za trpělivé odpovědi na mé dotazy, cenné rady, motivaci a podporu při jejím vypracovávání.

Obsah

Kapitola 1

Úvod

V dnešní době se terabajty stávají běžnou jednotkou kapacity, přenosová rychlost sítí stoupá na stovky gigabitů za sekundu a objem dat mnohonásobně převyšuje výkon počítačů a jejich schopnost data v krátkém časovém úseku zpracovat. Výsledky vědeckých a medicínských výpočtů dosahují až jednotek petabajtů, které je potřeba přesunout a posléze interpretovat. Obecně přenos velkého objemu dat po síti často představuje zásadní zpomalení celého procesu, kterého je nezbytnou součástí. Klade se proto velký důraz na zvyšování rychlosti směrovačů a jednotlivých linek.

Zrychlení zpracování nebo klasifikace paketů lze dosáhnout hardwarovou akcelerací. Právě pro tento účel slouží síťové karty obsahující programovatelné hradlové pole, neboli FPGA čip. Tento čip je možné využít k akceleraci již velmi rychlého kompresního algoritmu, a tím získat datovou kompresi tzv. on-the-fly, tedy pracující v reálném čase. Takto je možné ušetřit značnou část přenosového pásma.

V rámci této práce je velmi rychlá kompresní metoda LZ4 nejprve implementována v jazyce C. Následně je tato softwarová verze převedena do prostředí Catapult a omezeného jazyka C++, který je v prostředí dostupný. Pomocí nástroje Catapult a high-level syntézy je převedená verze verifikována oproti softwarové a automatizovaně převedena do VHDL kódu. Jednotlivé implementace jsou porovnány a na základě kompresního poměru a rychlosti komprese a dekomprese obvodového řešení vyvozeny závěry. Práce je také příkladem demonstrujícím výhody a nevýhody a pohodlí návrhu a vývoje pomocí high-level syntézy místo klasického přístupu s VHDL.

Tato práce obsahuje následující kapitoly. Kapitola 2 se věnuje vysvětlení pojmů z oblasti komprese, představuje základní druhy komprese různých typů dat a některé přístupy porovnává. V kapitole 3 je podobně popsaná specifikace algoritmu LZ4. Kapitola 4 předkládá omezení FPGA platformy, na které je nutné pamatovat, prostředí, ve kterém budou výsledné programy operovat, a důsledky z toho vyplývající. Dále obsahuje rozbor možných řešení a návrh kompresního a dekompresního modulu. Kapitola 5 popisuje postup implementace obou modulů v softwarové i hardwarové podobě a aplikované optimalizace. Srovnání jednotlivých implementací kompresorů a dekompresorů, rozbor jejich parametrů a návrhy na další optimalizace se nachází v kapitole 6. Kapitola 7 shrnuje dosažené výsledky a omezení použití high-level syntézy.

Kapitola 2

Pojmy a základy problematiky datové komprese

Pod kompresí zprávy, či obecně dat, chápeme jejich zkrácení nebo zmenšení na co nejnižší počet bitů, aniž by obsažené informace byly fundamentálně narušeny (změněny). Tohoto lze docílit pomocí odstranění přebytečných (redundantních) částí, které nejsou pro zachování původního významu dat kritické. Jedná se tedy o proces překódování vstupního toku dat na tok jiný, pokud možno s menší velikostí. Tato kapitola je založena na publikaci [12].

2.1 Entropie

Objem dat, který je možno odebrat, má svůj limit. Všechna data obsahují jisté množství informace, které je nepřímo úměrné pravděpodobnosti výskytu těchto dat. Množství informace I obsažené v datech D udává rovnice (2.1), kde $P(D)$ je pravděpodobnost výskytu dat D.

$$I(D) = \log\left(\frac{1}{P(D)}\right) = -\log P(D) \tag{2.1}$$

Jelikož jsou data reprezentována binárně, logaritmus je o základu 2, s čímž je spojená i jednotka množství informace *shannon*. Výsledný vztah, který je ekvivalentní sumě množství informace všech symbolů dat, lze zapsat ve tvaru rovnice (2.2), kde $P(d_i)$ je pravděpodobnost výskytu symbolu d_i v datech D a $|D| = n$ je počet symbolů.

$$I(D) = -\log_2 P(D) = \sum_{i=1}^{n} -\log_2 P(d_i) \quad [\mathrm{Sh}] \tag{2.2}$$

Tento vztah platí, pokud jsou symboly na sobě statisticky nezávislé. Potom se tato data označují jako zdroj dat bez paměti. Tento zdroj je možné popsat abecedou symbolů S a jejich pravděpodobnostmi výskytů $P(s_i)$, kde $s_i \in S$.

Již víme, že množství informace lze vypočítat i pro jednotlivé symboly podle vztahu (2.2). Pravděpodobnost, že symbol s_i bude množství informace $I(s_i)$ nést, je rovna právě $P(s_i)$, tedy pravděpodobnosti výskytu samotného symbolu. Na základě těchto znalostí můžeme zjistit průměrné množství informace na symbol, tak jak udává rovnice (2.3), což se nazývá entropie $H(S)$ abecedy S daného zdroje dat bez paměti.

$$H(S) = \sum_{S} P(s_i) \cdot I(s_i) = -\sum_{S} P(s_i) \cdot \log_2 P(s_i) \quad \left[\frac{\text{Sh}}{\text{symbol}}\right] \tag{2.3}$$

Komprese dat, tedy zahazování redundantních částí, je limitována právě entropií zdroje. Informace jsou pak (jak daný algoritmus dovoluje) maximálně zhuštěné. Tímto se však komprimovaná data stávají velice citlivými, co se týče chyb. Jakékoliv porušení sebemenší části může vést k jejímu nevratnému fundamentálnímu poškození, které nelze napravit. Princip komprese je duální se zabezpečováním dat proti chybám – komprese redundanci redukuje, přičemž zabezpečování dat redundanci naopak přidává. Právě díky přidání dodatečných informací je možné určité procento chyb opravit.

Ve většině dat, která nějakou reálnou informaci reprezentují, je možné nalézt mezi jednotlivými symboly závislosti. Tato data jsou označována jako zdroj dat s pamětí. Nejlepším příkladem takového zdroje je text. Symbol může představovat písmeno, slovo, slovní spojení, větu či celé kusy textu. Budeme-li uvažovat jednotlivá písmena jako symboly, lze závislost ukázat na digramech nebo trigramech, tedy mezi dvojicemi popř. trojicemi za sebou jdoucích písmen. Často používané digramy závisí na zvoleném jazyce. Vezmeme-li angličtinu, jeden z častých digramů je např. „th“. Symbol „h“ má určitou pravděpodobnost výskytu. Určíme-li však jeho pravděpodobnost výskytu bezprostředně za symbolem „t“ vzhledem k anglickému jazyku, bude tato pravděpodobnost mnohem vyšší, a tedy entropie je nižší, což umožnuje odstranit více očekávaných, tudíž nepotřebných, dat. A právě tyto statistické závislosti vedou k efektivnější kompresi.

Pomocí frekvenční analýzy symbolů ve zdroji dat je možné určit kompresní potenciál. Čím více se frekvenční graf blíží uniformnímu rozložení pravděpodobnosti, tím více entropie data obsahují a tím hůře se zároveň komprimují. Pokud mají všechny symboly stejnou frekvenci výskytu, nemají mezi sebou žádnou vazbu a lze tato data označit za náhodná – komprese není možná. Dalším příkladem nekomprimovatelných dat jsou již komprimovaná data. Jelikož většina (a pokud možno všechna) redundance již byla odstraněna, nelze žádné další komponenty symbolů vynechat, a tudíž zmenšit velikost.

2.2 Kompresor a dekompresor

Proces komprese je obecně asymetrický. Je nutno mít k dispozici implementaci dvou komponent, a to kompresoru a dekompresoru. Kompresor na vstupu přijímá data a produkuje na výstup data zkomprimovaná. Dekompresor provádí proces opačný – ze vstupních komprimovaných dat vytváří data v původní podobě. To je však nutno brát s rezervou, neboť ne vždy budou dekomprimovaná data identická s původními daty, viz 2.4.1.

Dvojice kompresor a dekompresor se také nazývá kodér a dekodér. Jelikož se data transformují ze vstupního kódu na cílový kód a naopak, je kompresi možno klasifikovat jako druh kódování, při kterém je reprezentace dat převedena do efektivní podoby.

2.3 Kompresní poměr a faktor

Aby bylo možné existující kompresní techniky porovnávat, je nutné vytvořit prostředky umožňující jednotlivé algoritmy kvalitativně ohodnotit jednotným způsobem. Pro tento účel existuje kompresní výkonnost [11]. Jedná se o sadu metrik k měření určitých parametrů, které jsou všemi kompresními metodami sdílené.

Jedna z nejpoužívanějších metrik je bezpochyby kompresní poměr definovaný rovnicí (2.4), která vyjadřuje procentuální velikost výsledných komprimovaných dat vůči původním vstupním datům.

$$\text{kompresní poměr} = \frac{\text{velikost výstupních dat}}{\text{velikost vstupních dat}} \cdot 100 \quad [\%] \tag{2.4}$$

Čím menší je kompresní poměr, tím lépe umí daný algoritmus redundanci redukovat. V případě, že hodnota poměru je větší než 100 %, tedy výstupní data jsou větší než vstupní, lze předpokládat, že kompresor není schopen detekovat a odstranit žádné nadbytečné části dat a pouze přidal svá interní metadata. Dochází zde k tzv. expanzi. Příčinou této situace může být nevhodně navržený kompresní algoritmus, nebo zdrojová data, jejichž entropie je natolik vysoká, že je shodná s entropií náhodných dat. Je zřejmé, že tato metrika (stejně tak jako ostatní metriky udávající kompresní výkonnost) je nezávislá na použité metodě, neboť ať už je jádro postavené na jakémkoliv principu, komprimovaný výstup bude generovat vždy.

Jednu z dalších metrik představuje kompresní faktor definovaný rovnicí (2.5), který je převrácenou hodnotou kompresního poměru.

$$\text{kompresní faktor} = \frac{\text{velikost vstupních dat}}{\text{velikost výstupních dat}} \tag{2.5}$$

Pro mnohé je tento vztah přirozenější než předchozí. Vyjadřuje, kolikanásobně jsou výstupní data menší než původní data na vstupu. K expanzi dochází, pokud je výsledná hodnota menší než 1.

Pokud u vztahu (2.4) pro kompresní poměr vynecháme konečné vynásobení, lze získat zajímavou statistickou veličinu, a to kolik výstupních jednotek je potřeba pro zakódování jedné vstupní. Záleží však, v jakých jednotkách byl výpočet proveden. Např. pokud by do vztahu byly dosazeny velikosti v bajtech, výsledkem by byl průměrný počet bajtů ve výsledných datech na jeden bajt vstupní. V případě vyjádření velikostí v bitech lze o veličině uvažovat jako o bitrate. Obecným cílem komprese je při kódování používat co nejnižší bitrate.

2.4 Základní druhy komprese

Zatím neexistuje žádná efektivní univerzální kompresní metoda. Setkáváme se s mnoha různými typy dat. Mezi základní patří multimédia (obrázky, videa, zvukové stopy), textové dokumenty (obyčejný text, číselné tabulky nebo i speciálně formátované dokumenty pomocí značkovacích jazyků jako je XML) a binární data. Každý typ má své specifické rysy, kterými se význačně odlišuje od ostatních (odlišnosti existují i v rámci stejné kategorie, jako v případě obrázků a zvuku). Proto bylo vytvořeno tolik kompresních metod, neboť mají-li být efektivní, je nutné se specializovat na určitý typ dat a plně využít charakteristik nacházejících se v něm.

Obecně lze jednotlivé metody rozdělit do několika základních skupin podle jejich cílů a aplikovaných principů [11]. Mnoho skupin se však kombinuje a prolíná. Existuje i řada dalších druhů komprese kromě zde uvedených, jako např. perceptivní komprese, která však není předmětem této práce.

2.4.1 Ztrátová a bezeztrátová komprese

Po procesu komprese je požadováno, aby data po dekompresi byla ve stejné podobě jako původní data, resp. aby výsledná interpretovaná informace nebyla nijak narušena či pozměněna. Pokud by např. v textovém souboru byla jiná slova, nesmyslné věty nebo by kusy textu chyběly důsledkem komprese, nebyla by taková metoda reálně použitelná. Proto se často uplatňují bezeztrátové kompresní metody, jejichž dekompresor generuje identická data s originálními.

Existují však výjimky, u kterých i přes určitou úroveň ztráty jsou data schopna si zachovat podstatnou část informací, resp. informace zpracovatelné člověkem. Příkladem může být jedna forma textové komprese, i když není z praktického hlediska vhodná. Pokud bychom z textu odstranili všechny mezery (obecně značeno jako bílé znaky nebo *white spaces*) a všechna počáteční písmena slov změnili na velké, dosáhli bychom jistého, avšak celkem zanedbatelně výhodného, kompresního poměru a zároveň by informace zůstaly zachované. Data by se pouze komprimovala, neboť pokud by dekompresor vykonával proces opačný, mohlo by dojít k odstranění některých detailů. Důvodem by byla neschopnost dekompresoru lidsky uvažovat, což by mělo za následek odstranění některých velkých písmen, která by správně velká zůstat měla. Jelikož je člověk schopen kontextově rozlišit běžná slova od názvů či jmen, dokáže se s přebytečnými velkými písmeny vyrovnat. Tuto metodu je možné považovat za příklad ztrátové komprese – kompresní poměr bude menší než 100 % a podstatná část dat je člověkem stále interpretovatelná.

Reálně se však ztrátová komprese používá v oblasti multimédií, kde se využívá nedokonalostí lidských vjemů za použití speciálních technik.

2.4.2 Dvouprůchodová komprese

Některé algoritmy potřebují pro své korektní a požadované fungování sbírat jisté statistiky o vstupních datech. Ty pak využívají k efektivnější kompresi, neboť předem disponují informacemi o skladbě vstupu. Klasický postup těchto metod se tedy skládá ze dvou průchodů dat. První je analyzační, který slouží pouze k nasbírání nutných statistik, druhý průchod již vstupní data komprimuje na základě předešlých zjištěných informací.

2.4.3 Adaptivní a neadaptivní komprese

Neadaptivní metody se nijak kromě typu dat, jak je uvedeno výše, nepřizpůsobují. Jejich operace a parametry zůstávají neměnné po celou dobu fungování. Adaptivní kompresory se chovají naopak. Dvouprůchodové metody jsou v podstatě podtřídou adaptivních metod. Obě třídy přizpůsobují své parametry a popř. i operace na míru vstupu daného typu dle jeho charakteristik. Dvouprůchodové kompresory svou činnost provádí sériově – analýza a poté komprese. Tímto je zaručeno optimální nastavení parametrů algoritmu. U ostatních adaptivních metod dochází k těmto procesům paralelně – kompresor data komprimuje a zároveň sbírá statistiky, podle kterých záhy algoritmus přizpůsobuje. Přirozeně tento způsob vede k zásadně kratší době komprese dat. Nejedná se však pouze o vylepšení, nýbrž o kompromis. Jelikož kompresor nemá o vstupu k dispozici všechny dostupné informace, parametry algoritmu nejsou optimální. Jedná se o postupně se zpřesňující aproximaci optimálních parametrů. Je zde zřejmý kompromis mezi efektivnějším kompresním poměrem a rychlostí procesu komprese. Obě varianty nachází své uplatnění, dvouprůchodové metody jsou používané hlavně pro archivační účely.

2.4.4 Proudová a bloková komprese

Většina kompresních algoritmů pracuje v proudovém režimu, ve kterém jsou vstupní data postupně po malých částech načítána a zpracovávána, dokud nejsou veškerá data zpracována. LZ4 se právě řadí mezi proudové kompresní algoritmy. Metody jako Burrows-Wheelerova, operující v blokovém režimu, potřebují vždy konstantní blok dat, který je posléze samostatně zpracován, podobně jako v případě ECB (electronic code book) režimu symetrických blokových šifer. Volba vhodné velikosti pracovního bloku má přímý vliv na kompresní výkonnost dané metody.

2.4.5 Časově symetrická a asymetrická komprese

Klasifikaci dle symetrie lze aplikovat i na časovou náročnost vykonávaných procesů. Kompresní metody pro běžné používání mají zpravidla vyrovnané doby trvání komprese i dekomprese, neboť obě funkce jsou využívány stejně často. Takové metody jsou tedy časově symetrické. V případě časově asymetrických metod jedna z funkcí trvá mnohem déle oproti druhé, neboť vykonává podstatně složitější algoritmus. Komplexní kompresor a jednoduchý dekompresor jsou ideální pro případy, kdy jsou data často používaná, ale jejich obsah se mění pouze zřídka. Příkladem mohou být přenosná média, která jsou pouze pro čtení (CD, DVD), iPXE firmware pro zavedení systému ze sítě [2], jádra operačních systémů stahovaná z TFTP serveru a masově nasazované aktualizace programů. Opačný případ, tedy rychlá komprese a zdlouhavá dekomprese, připadá v úvahu při častých změnách cílových dat. Tento scénář lze pozorovat např. při periodickém zálohování dat.

LZ4 se snaží být co nejrychlejší v obou případech použití, proto patří k časově symetrickým metodám.

2.5 Základní principy kompresních algoritmů

Existuje řada technik [10], jak komprese dosáhnout. V počátcích rozvoje výpočetní techniky byly jednotlivé přístupy dostačující. V dnešní době však už samostatně nedosahují uspokojivých výsledků, a proto se tyto principy velmi často kombinují ve snaze zvýšit výslednou výkonnost metody.

2.5.1 Run-Length kódování

Ideou metody kódování Run-Length (také označované jako RLE) je nahrazování n výskytů nějakého symbolu s jedním párem ns. n po sobě jdoucích identických symbolů se anglicky nazývá *run-length* délky n, odtud také pochází použitý název principu. Do češtiny by bylo termín možné přeložit jako běh znaků. Tento postup lze obecně aplikovat ve více oblastech.

2.5.2 Diferencování

Tento princip, také označován jako relativní kódování, který je možné použít v případě, že vstupními daty jsou podobné textové řetězce nebo řetězce čísel, jejichž hodnoty se příliš neliší. Nejčastější uplatnění se nachází v telemetrii, kde senzorová zařízení sbírají v určitých časových intervalech data jako např. teplotu, vlhkost vzduchu a atmosférický tlak a posílají je k dalšímu zpracování v rámci senzorové sítě, průmyslového řídicího systému, inteligentní budovy či domácnosti a jiných aplikací. Sledování hodnot probíhá v pravidelných krátkých intervalech. Rozdíl po sobě jdoucích hodnot tedy bude ve většině případů malý. Počáteční

číslo sekvence musí být vždy přeneseno plnou hodnotou (verbatim), následující hodnoty jsou již kódovány jako diference od hodnoty předchozí. V případě příliš velké skokové změny, kdy diference není schopna takový rozdíl popsat, je nutné poslat hodnotu opět celou.

2.5.3 Statistické metody

Nejznámější, nejrozšířenější a zároveň velmi efektivní statistickou metodou je Huffmanovo kódování, které symbolům přiřazuje kód s proměnnou délkou podle jejich frekvence výskytu. Ideální střední délka kódu, která se rovná entropii vstupních dat, je ale generována pouze v případě, že pravděpodobnost (zde zaměnitelná s frekvencí) výskytů jednotlivých symbolů je rovna záporným mocninám dvou. V ostatních případech je ideální bitová délka kódů neceločíselná, což reálně není proveditelné. Vždy je nutné délku kódu zaokrouhlit na celé bity a právě zde vzniká neefektivita. Pokud je pravděpodobnost symbolu a rovna např. $0,46$, množství informace, jež symbol nese, je podle vztahu 2.2 $I(a) = -\log_2 P(a) \doteq 1{,}12\,Sh$, což i přesně odpovídá ideální bitové délce přiděleného kódu. Huffmanovým kódováním tak zde ztratíme skoro celý bit.

2.5.4 Aritmetické kódování

Aritmetické kódování se snaží zmiňovanou neefektivitu statistických metod odstranit. Kódy nejsou přidělovány jednotlivým symbolům, nýbrž celý datový vstup má přiřazený jeden velmi dlouhý kód. Jedná se o dvouprůchodovou metodu, neboť je potřeba znát pravděpodobnosti výskytů symbolů ve vstupních datech, které se zjistí analyzačním průchodem daty před samotnou kompresí. Pravděpodobnosti lze však také aproximovat jinými způsoby nebo od jiných zdrojů, a tím analyzační krok přeskočit. Na začátku komprese je stanoven interval $[0, 1)$. Princip samotné komprese spočívá v zužování uvedeného intervalu při čtení vstupních symbolů proporcionálně k jejich pravděpodobnostem. Tímto zároveň roste počet desetinných míst hranicím intervalu, tudíž bude pro jejich reprezentaci nutno více bitů. Čím vyšší má symbol pravděpodobnost, o to méně je interval zúžen a tím méně bity ve výsledku přispěje.

2.5.5 Komprese s využitím slovníku

Statistické kompresní metody sbírají informace o vstupních datech a vytvářejí si tak jejich model. Výsledná kvalita komprese, tedy velikost kompresního poměru, silně závisí na přesnosti sestrojovaného modelu. Metody založené na slovníku žádný statistický model dat nevytváří a nepoužívají ani kódování s proměnnou délkou. Nekódují se jednotlivé symboly, ale celé řetězce symbolů pomocí tzv. slovníkového tokenu, což umožňuje těmto metodám (v případě velkého objemu vstupních dat) data zkomprimovat až na úroveň jejich entropie. Lze je tedy označovat jako entropické kodéry. V běžných situacích dosahují výkonnosti, která slovníkové metody učinila velmi populárními – nachází se u komprese textu, obrázků a dokonce i audia.

Slovník může být statický nebo dynamický. Statický slovník je stálý, občas dovoluje vkládání nových řetězců, ale žádné nelze mazat. Nejjednodušším příkladem je slovník obsahující různá česká slova. Slovníkovým tokenem by zde byl index (neboli pořadí) jednotlivých slov. Ze vstupu jsou načítána slova, tedy znaky oddělené mezerami či interpunkcí, která jsou poté vyhledávána ve slovníku. Je-li slovo nalezeno, na výstup je vypsán index tohoto slova ve slovníku. V opačném případě, pokud se dané slovo ve slovníku nenachází, musí být na výstup zapsáno celé slovo v původní podobě. Tak jak tomu bylo u předchozích principů,

opět zde nastává míchání dvou typů datových položek – indexů a nekomprimovaných slov. Možné řešení představuje dodatečný flag bit, jako v případě diferencování.

Nechť slovník obsahuje 32768 položek. K naindexování těchto 2^{15} položek je tedy potřeba 15 bitů. Přidáme-li další 16. bit, lze index od slova jednoduše a hlavně jednoznačně odlišit. Dekodér by vždy načítal 2 bajty. Podle prvního bitu by určil, zda se jedná o index. Pokud ano, dekodér zaadresuje slovník 15bitovým indexem a uložené slovo vypíše na výstup. Pokud se o index nejedná, zbylých 15 bitů určuje délku nekomprimovaného slova na vstupu v bajtech okamžitě následujícího za načtenými dvěma bajty.

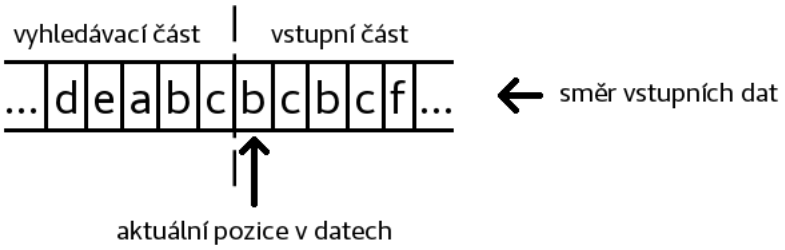
Pokud je zvolen dostatečně velký slovník, u textů napsaných v přirozeném jazyce dosáhneme uspokojivého kompresního poměru. Ostatní textová data, jako jsou např. zdrojové kódy programů, však obsahují slova, která v žádném přirozeném jazyce nejsou vlastní. U binárních dat je situace mnohem horší, neboť v nich se málo kdy vyskytují vůbec nějaká slova. Místo komprese bychom tedy spolehlivě dosáhli expanze dat. Proto nejsou statické slovníky pro obecné použití vhodné a uplatnění nachází spíše ve specifických aplikacích. Např. posílá-li se mezi dvěma uzly pouze jeden typ dat, lze vytvořit speciální slovník obsahující řetězce nejčastěji se vyskytující v zasílaných datech. Tímto lze snadno zefektivnit přenos po datových linkách spojující zmiňované dva uzly.

Druhá varianta slovníku, dynamická (také označovaná jako adaptivní), umožňuje mazat řetězce a vkládat nové, tak jak jsou nacházeny čtením vstupních dat. Tento typ slovníku je pro obecné použití preferovaný. Komprese začíná s prázdným slovníkem nebo může obsahovat několik výchozích záznamů. Zpracovávaná slova jsou postupně do slovníku přidávána. Slovník však může mít stanovenou maximální velikost – v případě jeho saturace nebudou nové záznamy pouze vkládány, nýbrž budou nahrazovat již existující. Výběr tzv. oběti k odstranění ze slovníku může být náhodný nebo určený frekvencí jeho užívání, neboť velmi málo používaný záznam není užitečný a zabírá místo potenciálně frekventovanému řetězci. Příliš rozsáhlé slovníky znamenají dlouhé vyhledávací časy. Je tedy nutné najít kompromis mezi rychlostí metody a výslednou efektivitou. Dekodér postupně skládá slovník stejným způsobem jako kodér. Pokud kompresor narazí na nový řetězec, je nutné, aby nejdříve řetězec vypsal v nekomprimované podobě na výstup a až pak ho vložil do slovníku. Jinak by dekompresor mohl narazit na index, který v jeho slovníku není obsazen, a nebyl by tak schopen data dekódovat. Některé metody po zaplnění slovníku již nové záznamy nevkládají, jiné zavádějí operace jako kompletní vymazání obsahu slovníku.

Kapitola 3

Kompresní metoda LZ4

Kompresní algoritmus LZ4 [3] patří do kategorie bezeztrátových slovníkových metod. Vychází z poměrně starého algoritmu LZ77, se kterým sdílí mnoho rysů. Nejdůležitějším aspektem je tzv. *sliding window* neboli klouzavé okno, ilustrované na obrázku 3.1. Jedná se o výseč určité velikosti ve vstupních datech, která se postupně daty pohybuje (klouže po nich). Okno se skládá ze dvou částí – z části vstupní či kódovací a z části vyhledávací. Vstupní část obsahuje data, která mají být zkomprimována. Vyhledávací část okna je jistým slovníkem této metody, neboť obsahuje část již zpracovaných dat okamžitě se nacházejících za ještě nezpracovanými daty a v nich probíhá hledání sekvencí bajtů z aktuální vstupní části. Komprese se dosáhne odkazováním dřívějších sekvencí bajtů pomocí indexu, který určuje relativní pozici začátku sekvence od momentálního konce dekomprimovaných dat, a délky této sekvence. Dřívější identické části dat jsou tedy pouze kopírovány z předchozích výskytů. Velkou výhodou je flexibilita vyhledávání sekvencí, neboť není pevně předepsaný žádný algoritmus. Lze tak vytvořit mnoho variant s různými parametry a na základě požadavků zvolit vyhovující.



Obrázek 3.1: Ukázka *sliding window* z algoritmu LZ77

3.1 LZ4 rámec

Veškerá data jsou zabalená v tzv. rámci, jehož strukturu znázorňuje obrazek 3.2. Celý vstup bývá většinou pouze v jednom rámci, těch však může být za sebou libovolný počet. Specifikace nijak neurčuje chování v případě existence více rámců, jedná se tedy o případné

rozšíření dané implementace. Referenční implementace[1] v takové situaci rámce zpracuje v sekvenčním pořadí.

Obrázek 3.2: Struktura LZ4 rámce

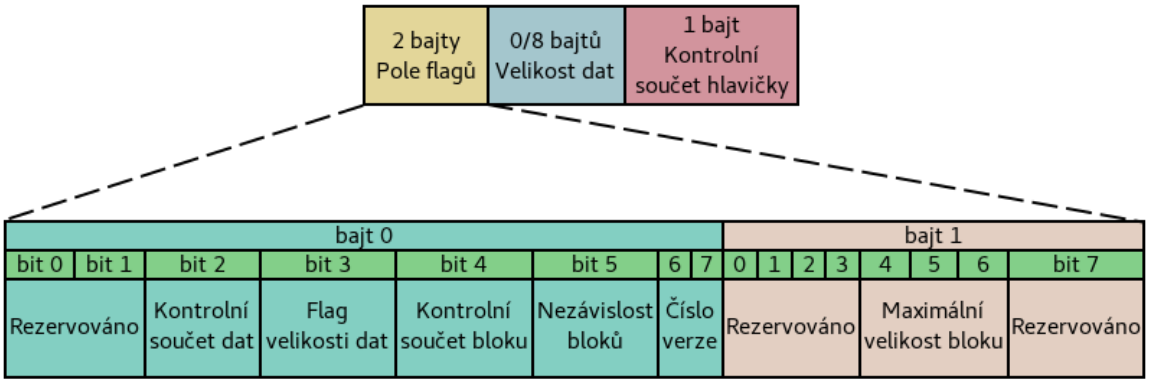
- *Magická konstanta:* unikátní hodnota identifikující LZ4 datové rámce dané verze. Aktuální verze 1.5.0 používá konstantu `0x184D2204` hexadecimálně. Správce souborů a jiné programy, které zobrazují typ souborů, tak mohou snadno LZ4 archiv rozpoznat.
- *Rámcový deskriptor:* popisuje vlastnosti daného rámce. Skládá se celkem ze tří částí, a to z pole tzv. flagů, velikosti dat a kontrolního součtu deskriptoru, viz sekce 3.2.
- *Blok:* obsahuje samotná data, detailní popis viz sekce 3.3.
- *Koncová značka:* 4 nulové bajty značící konec sekvence datových bloků. Jedná se v podstatě o velikost následujícího bloku.
- *Kontrolní součet dat:* zabezpečuje vstupní data proti chybám. K výpočtu je použita hash funkce xxHash[2] od stejného autora jako LZ4, která jako vstup přijímá veškerá data ke kompresi. Inicializační *seed* parametr je nastaven na nulu. Tato položka není povinná, nýbrž doporučená, a je přítomna, pokud je nastaven příslušný flag v deskriptoru.

3.2 Rámcový deskriptor

Deskriptor, který je rozkreslený na obrázku 3.3, uchovává nastavené parametry LZ4 rámce, popř. velikost původních vstupních dat. Popisuje, které volitelné části se v rámci nacházejí, určuje maximální velikost extrahovaných dat z jednotlivých datových bloků a označuje verzi formátu samotného rámce. Zaujímá minimálně 3 bajty, obsahuje-li volitelné části, tak 11 bajtů.
Pole flagů se skládá z následujících položek:

- *Kontrolní součet dat:* v případě nastaveného flagu se na konci rámce za koncovou značkou nachází čtyřbajtový kontrolní součet původních dat, kterým lze zachytit narušení dat chybnou kompresí nebo přenosem.
- *Flag velikosti dat:* je-li tato položka nastavena, za polem flagů se v deskriptoru nachází velikost původních nezkomprimovaných dat ve formě osmi bajtů zapsaná neznaménkově ve formátu little endian. V opačném případě je velikost dat vynechána.
- *Kontrolní součet bloku:* určuje přítomnost kontrolního součtu o velikosti čtyř bajtů na konci každého datového bloku. Slouží k detekci případné chyby v bloku ještě před samotným dekódováním.

[1] https://github.com/lz4/lz4
[2] Velmi rychlá nekryptografická hash funkce, která se rychlostí blíží limitům operační paměti. Dostupná z http://cyan4973.github.io/xxHash/.



Obrázek 3.3: Struktura rámcového deskriptoru, kde bit i bajt 0 je nejnižší

- *Nezávislost bloků:* pokud je tento flag nastaven na 1, jednotlivé bloky jsou na sobě nezávislé, a proto je možné bloky dekódovat paralelně, což umožňuje vícevláknové zpracování. Každý blok má svoje vlastní vyhledávací okno. Je-li flag 0, každý následující blok je závislý na předchozím (přesněji řečeno na posledních 64 KiB[3] dat předchozího bloku) a je nutné všechny bloky dekódovat sekvenčně. Všechny bloky sdílí jedno vyhledávací okno.

- *Číslo verze:* dvoubitové číslo, které musí být nastavené na 1, binárně `0b01`. Pokud je deskriptor jiné verze, dekodér zde popisované verze nemůže takový rámec dekódovat, neboť deskriptory různých verzí mohou mít odlišné položky.

- *Maximální velikost bloku:* má za účel u paměťově omezených aplikací dopředu dekodéru sdělit, kolik lze maximálně očekávat dekomprimovaných dat z jednoho datového bloku, aby mohl alokovat příslušné množství paměti. Dekodér může odmítnout zpracování rámce, který přesáhne určitou maximální velikost dekomprimovaných dat v bloku. Podporované velikosti jsou popsány v tabulce 3.1.

Hodnota max. velikosti bloku	0	1	2	3	4	5	6	7
Max. velikost dat v bloku	–	–	–	–	64 KiB	256 KiB	1 MiB	4 MiB

Tabulka 3.1: Maximální velikost původních dat v bloku podle tříbitové hodnoty položky

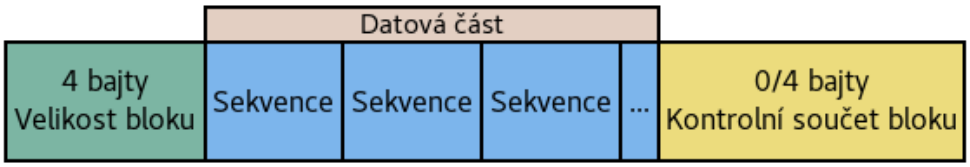
- Rezervované bity musí být nulové. Mohou být použity v budoucích verzích specifikace. V takovém případě by dekodér konformní k aktuální verzi neměl být schopen rámec dekódovat.

Následující položka *Velikost dat* určuje velikost původních dat v nekomprimované podobě. Velikost je přítomna v deskriptoru, pouze pokud je nastavený příslušný flag velikosti dat v poli flagů. *Kontrolní součet hlavičky* obsahuje kontrolní součet celého deskriptoru včetně volitelné velikosti dat, je-li přítomna. K jeho výpočtu je opět použita hash funkce xxHash s nulovým inicializačním *seed* parametrem, z jejíhož výsledku se využije druhý nejnižší bajt. Tento kontrolní součet je povinný.

[3] 1 MiB (mebibajt) = 1024 KiB (kibibajtů) = 1024^2 B (bajtů)

3.3 Datový blok

Datový blok obsahuje již samotná data. Jak je možné vidět na obrázku 3.4, na začátku každého bloku je udána jeho velikost, za kterou se nachází buď data nekomprimovaná nebo komprimovaná v podobě za sebou jdoucích kompresních sekvencí. Sekvence mají zpravidla jednotnou strukturu, poslední sekvence je však výjimkou. Na konci bloku se může nacházet kontrolní součet daného datového bloku.



Obrázek 3.4: Struktura datového bloku

- *Velikost bloku:* udává velikost následující datové části a nezahrnuje kontrolní součet, pokud je přítomen. Velikost je zapsána ve formátu little endian. Nejvyšší bit má však jinou funkci – je-li nulový, datová část se skládá ze sekvencí zkomprimovaných dat, v opačném případě se zde nachází nekomprimovaná část původních dat. Velikost nesmí přesáhnout maximální velikost bloku určenou v deskriptoru bez ohledu na typ bloku. Taková situace by mohla nastat v případě, že jsou data nekomprimovatelná a přidáním LZ4 specifických metadat dojde k expanzi. Proto musí být taková data obsažena v nekomprimovaném bloku.

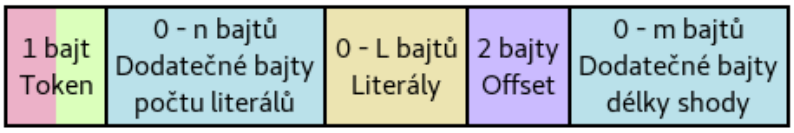
- *Datová část:* obsahuje vlastní data. Může se jednat o nekomprimovaná data, která lze bez dalšího zpracování rovnou poslat na výstup, nebo se zde vyskytuje posloupnost kompresních sekvencí, ze kterých se data extrahují.

- *Kontrolní součet bloku:* čtyřbajtový kontrolní součet vypočítaný z datové části bloku xxHash funkcí s parametrem *seed* nastaveným na nulu. Cílem je identifikovat poškozená data před samotným dekódováním dat. Tato položka se za datovou částí nachází pouze v případě nastavení příslušného flagu v rámcovém deskriptoru, není tedy povinná.

3.4 Kompresní sekvence

Samotné jádro komprese představuje posloupnost sekvencí. Specifikace popisuje pouze formát sekvence, nikoliv jak ji vytvořit. Takto postupuje řada moderních standardizovaných kompresních metod, především kodeky v oblasti videa. Standard určuje tvar komprimovaného datového toku a jak jej dekódovat. Kodér není nijak předepsán, tedy způsob, jak komprimovaná data v daném formátu generovat, je čistě na autorovi kodéru, který může zdrojový kód uzavřít a vytvořit tak proprietární implementaci [10]. Struktura LZ4 sekvence je shrnuta na obrázku 3.5.

- *Token:* vyskytuje se na začátku každé sekvence a je rozdělen na 2 poloviny. Čtyři vyšší bity určují počet literálů, což jsou neupravené bajty původních dat, které je možné přesunout na výstup, resp. na konec vyhledávacího okna. Pokud je hodnota poloviny 0,



Obrázek 3.5: Struktura LZ4 sekvence

v sekvenci nejsou žádné literály. Je-li hodnota rovna 15, je potřeba načíst další bajty k získání celkového počtu. K této hodnotě jsou hodnoty následujících *dodatečných bajtů počtu literálů* postupně přičítány, dokud aktuálně načtený bajt není hodnotou menší než 255. Takový bajt je posledním přičteným a značí konec rozprostřeného počtu literálů.

Nižší čtyři bity stanovují délku shody. Existuje minimální délka shody, která není součástí hodnoty v tokenu, ale je k ní automaticky přičítána. Je-li tedy hodnota dolní poloviny tokenu 0, délka shody je rovna čtyřem. Pokud je rovna 15 (ve skutečnosti rovna 19), platí zde stejný mechanismus jako u počtu literálů. *Dodatečné bajty délky shody* se nachází na konci sekvence jako její poslední část.

- *Literály:* původní bajty v nekomprimované formě. Jejich počet závisí na délce získané z horních čtyř bitů *tokenu* a případně *dodatečných bajtů počtu literálů.*

- *Offset:* dvoubajtová hodnota ve formátu little endian udávající relativní pozici začátku shody od aktuálního konce vyhledávacího okna. Nulová hodnota offsetu není přípustná. Jelikož může být offset maximálně 65535, velikost vyhledávacího okna je 64 KiB.

Může nastat situace, kdy délka shody je větší než offset. Dochází tak ke kopírování nejen z předešlých dekomprimovaných dat, ale i z právě zkopírovaných, což je ilustrováno na obrázku 3.6. Tento případ se objevuje u opakující se posloupnosti bajtů.

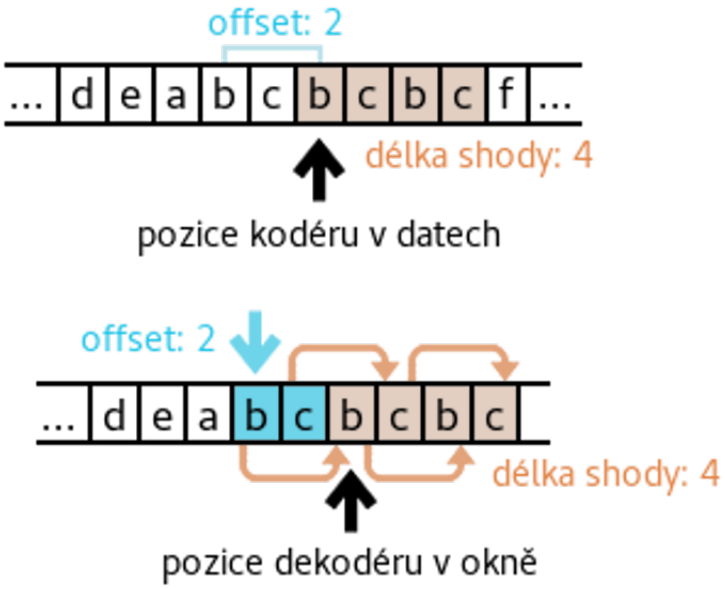
Má-li být implementovaný kodér kompatibilní s referenčním dekodérem, je nutné dodržet stanovená omezení (která ve specifikaci nejsou jednoznačným způsobem popsána):

1. Posledních 5 bajtů dat v každém komprimovaném bloku musí být v sekvenci uloženo jako literály, z čehož vyplývá, že poslední sekvence každého bloku není kompletní, ale končí literálovou částí.

2. Poslední shoda v bloku, tedy v předposlední sekvenci bloku, musí začínat minimálně 12 datových bajtů před koncem bloku, tudíž u souborů menších než 13 bajtů není možné dosáhnout komprese.

3.5 Uživatelský rámec

Kromě klasického rámce existuje i rámec uživatelský, také označovaný jako tzv. „přeskočitelný“, zobrazený na obrázku 3.7. Jedná se o rozšíření, které umožňuje vkládání aplikačně specifických dat mezi rámce. Obecný dekodér, který není schopen uživatelské rámce zpracovat, musí být schopen tyto rámce rychle přeskočit a pokračovat v dekompresi. Pomocí tohoto rámce je možné vylepšit metodu o schopnost komprimovat více souborů do jediného archivu. Pro každý komprimovaný soubor by byla vytvořena dvojice rámců – uživatelský by obsahoval vždy název původního souboru a následující klasický rámec by obsahoval



Obrázek 3.6: Ilustrace případu větší délky shody než offsetu

samotná data souboru. Kvůli správné identifikaci archivu magickou konstantou je doporučeno nezačínat uživatelským rámcem. V případě nutnosti lze před něj vložit datový rámec s jedním blokem nulové velikosti.

Obrázek 3.7: Struktura uživatelského rámce

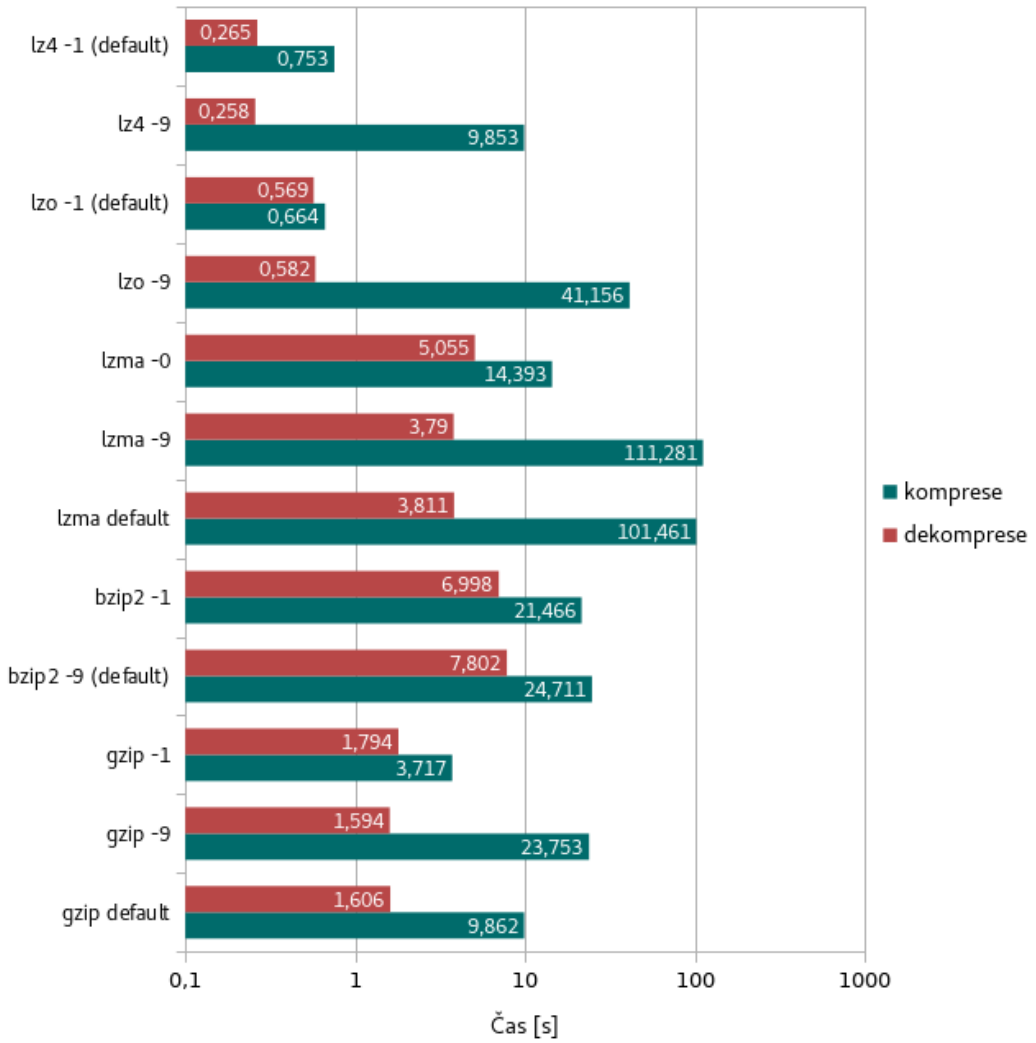
- *Magická konstanta:* uživatelský rámec může používat jednu z 16 konstant, hexadecimálně `0x184D2A5X`, kde `X` je volitelná hodnota. Aplikace tedy mohou mít až 16 typů specifických metadat.
- *Velikost dat rámce:* velikost samotných dat nacházejících se za touto položkou ve formátu little endian neznaménkově. Není do ní započítaná magická konstanta ani tato velikost.

3.6 Porovnání s ostatními kompresními algoritmy

Pro kvalitativní vyhodnocení parametrů kompresního algoritmu je vhodné provést porovnání. Byly zvoleny moderní kompresní programy jako gzip (deflate) a bzip2, které jsou běžně dostupné na linuxových distribucích, a dále LZMA a LZO – slovníkové metody, které patří do skupiny LZ algoritmů. Každá metoda byla testována ve výchozím (default) nastavení,

módu rychlé komprese (s přepínačem -0 či -1) a v módu nejlepší komprese (s přepínačem -9). V některých případech je jeden z extrémů nastaven právě jako výchozí mód.

Veškeré testy probíhaly jednovláknově na procesoru Intel Xeon E5-2680v3 @ 2.50GHz odděleného uzlu superpočítače Salomon[4]. Jako vstupní data slouží Silesia korpus[5], což je kolekce různých typů souborů převážně textového charakteru, která slouží právě k měření parametrů kompresních algoritmů. Obsahuje však i databáze, obrázky, binární data a spustitelné programy. Cílem tohoto korpusu je zahrnout nejpoužívanější moderní formáty dat. Jelikož je souhrnná velikost rovna 202 MiB, korpus je dostatečné velký na to, aby bylo možné změřit dobu vykonávání u rychlejších algoritmů. Metody však většinou neumí komprimovat více souborů do jednoho archivu a proto je nutné korpus spojit do jednoho souboru s využitím programu *tar*. Zdrojový i výsledný soubor se nacházel v RAM disku, což je virtuální disk mapovaný na operační paměť, čímž odpadá klasická disková režie a neovlivňuje tak samotné měření.

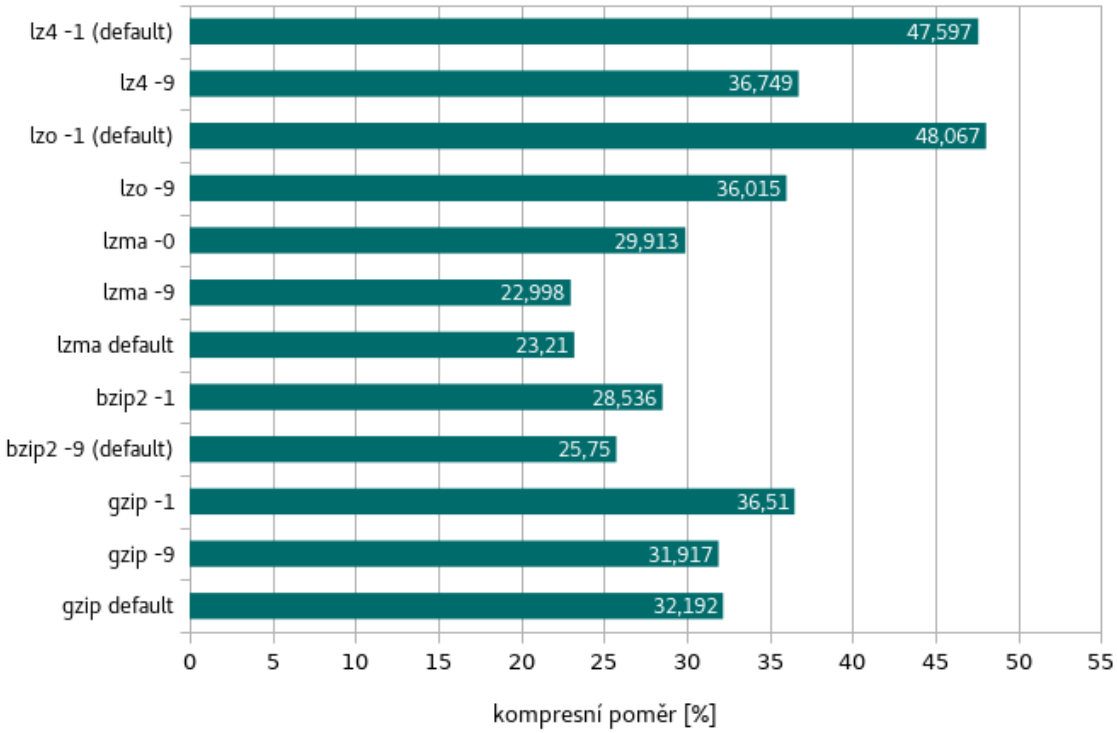


Obrázek 3.8: Graf dob vykonávání komprese a dekomprese zvolených algoritmů

[4] https://docs.it4i.cz/salomon/hardware-overview-1

[5] Dostupný na http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia

Z grafu 3.8 s horizontální osou v logaritmickém měřítku je zřejmé, že LZ4 disponuje nejrychlejší dekompresí ze všech testovaných algoritmů. Rychlostí komprese se metody LZ4 a LZO ve výchozím nastavení příliš neliší. Obě jsou velmi vyrovnané a mají potenciál pro použití v real-time aplikacích. Ostatní algoritmy potřebují mnohem více času pro kompresi i dekompresi ve všech módech. U LZ4 v módu nejlepší komprese rapidně naroste potřebný čas pro kompresi, neboť je nutné vytvářet rozsáhlé datové struktury. V tomto módu LZO již razantně zaostává za LZ4.

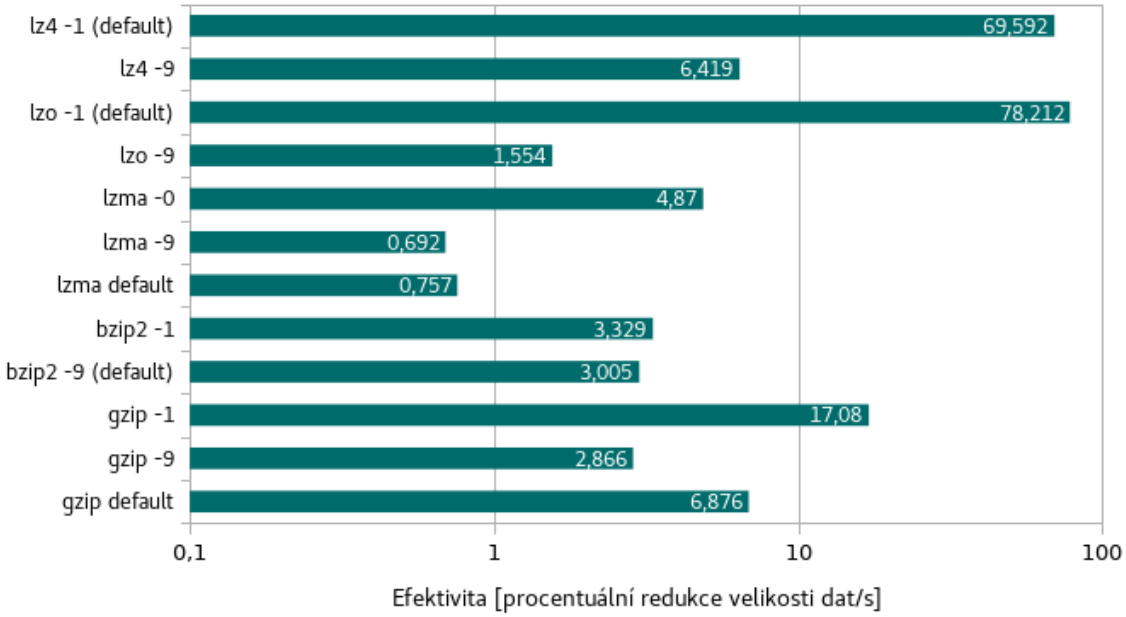


Obrázek 3.9: Graf výkonností zvolených algoritmů

LZ4 dosahuje krátké kompresní doby na úkor kompresního poměru, jak je možné vidět na grafu 3.9. I v oblasti kompresního poměru jsou si LZO a LZ4 velmi podobné. Nejvýkonnější metodou je LZMA, která potřebuje také odpovídající nejdelší dobu běhu. Uspokojivou alternativou k LZMA je bezpochyby bzip2, neboť dosahuje velmi podobného kompresního poměru za skoro pětinu času. Aby bylo zřetelné, zda je dodatečný procesorový čas při kompresi přínosný, je vhodné vypočítat získané procento kompresního poměru za jednotku času.

Jak ukazuje graf 3.10 s horizontální osou v logaritmickém měřítku, metoda LZMA v módu nejlepší komprese není efektivní z hlediska časové náročnosti. bzip2 je vhodným náhradníkem, pokud je nízký objem dat nezbytným požadavkem. Je možné si všimnout malé ceny módu nejlepší komprese. Oproti módu nejrychlejší komprese je ztráta efektivity velmi nízká ve srovnání s ostatními algoritmy. Metoda deflate, tedy gzip, je správnou volbou pro běžné používání, neboť poskytuje kompromis mezi kompresním poměrem a dobou komprese, avšak pouze ve výchozím módu. Mód nejlepší komprese již není výhodný. V oblasti efektivity jasně dominují LZO a LZ4. Má-li být metoda využívána v reálném čase pro

kompresi i dekompresi, vítězí LZ4, neboť doba dekomprese je oproti LZO poloviční a přitom kompresní rychlost je téměř identická.



Obrázek 3.10: Graf časové efektivity algoritmů

Kapitola 4

Návrh

Komprese a dekomprese bude oddělena na dva samostatné programy. Implementace bude probíhat ve dvou fázích. Výsledkem první fáze bude čistě softwarová verze obou programů napsaná v jazyce C a běžící na klasických procesorech. Po vyladění této verze započne fáze druhá, a to konverze z jazyka C do Catapult C a následné využití high-level syntézy k vygenerování výsledné verze pro hardware ve VHDL.

FPGA platforma obvykle disponuje přímo na čipu mnohem menší pamětí než klasické počítače. Každý KiB je cenný, a proto je nutné paměťové nároky hlídat a cílový program patřičně uzpůsobit. Kompresor i dekompresor jsou ve finále určeny pro FPGA čipy Virtex-7 H580T, které se nachází na síťových kartách řady COMBO-100G [1]. Rodina COMBO karet se skládá z vývojových desek, které se specializují na vysokorychlostní zpracování síťových dat. Cílem programů je zprostředkovat transparentní kompresi pro aplikace a jejich data v reálném čase. Z hlediska ISO/OSI referenčního modelu [6] bude komprese situována pod aplikační vrstvou. Jelikož se jedná o specifickou aplikaci, bude možné množinu potřebných funkcí programů zredukovat. Funkce, kterými budou programy disponovat, budou plně konformní s oficiální specifikací LZ4 algoritmu a budou se držet stanovených omezení tak, aby byla implementace kompatibilní s referenčním řešením autora algoritmu. Ani jeden z programů nebude ovládaný uživatelem, proto lze odstranit veškerou režii spojenou se spouštěním z příkazové řádky jako je nastavování a kontrola zadaných parametrů. Odstranění podmíněných částí z této oblasti také zlepší plynulost programu. V softwarové části bude načítání a ukládání dat řešeno pomocí standardního vstupu a výstupu, tedy pomocí souborů. V hardwarové verzi tato varianta není použitelná a bude potřeba vytvořit jiné rozhraní s vnějším prostředím popsané v sekci 4.3, které bude mít v ideálním případě podobné vlastnosti jako soubor.

4.1 Dekompresor

Paměťové nároky limituje paměť přítomná přímo na čipu, což je v případě cílové FPGA platformy Virtex-7 H580T 4 230 KiB blokové RAM a 1 106 KiB distribuované RAM v LUT [14]. Lze tedy alokovat buffer pro načtení celého komprimovaného nebo nekomprimovaného bloku, který dosahuje maximální velikosti 4 MiB. Vyhledávací okno, což je v podstatě buffer pro dekódovaná data, ze kterého probíhá kopírování match sekvencí, musí být schopno uchovat přesně 65 535 bajtů, neboť offset začátku match sekvence je velký 2 bajty. Z hlediska paměťových zdrojů jsou vyjmenované struktury nejnáročnější. Ostatní proměnné se budou nacházet ve zbylých cca. 1 240 KiB, což je kapacita více než postačující.

Vstupní data, tedy LZ4 archivy, jsou proudového charakteru, přenášená mezi dvěma body. Veškeré nalezené uživatelské rámce budou v této implementaci ve výchozím chování přeskočeny. Schopnost interpretovat uživatelské rámce lze později podle potřeby doplnit. Datové rámce je možné chápat jako jednotlivé datové transakce, i když jsou součástí jednoho archivu. Dekompresor nemá žádné povědomí o struktuře dat nebo jejich metadatech, pouze o jejich pořadí. Proto bude dekompresor zpracovávat po sobě jdoucí rámce sekvenčně a všechna získaná data v tomto pořadí přeposílat na výstup. Jelikož budou rámce putovat skrz síť, přesněji řečeno přes spojovaný kanál zaručující spolehlivý přenos (s největší pravděpodobností TCP kanál), není nutné počítat kontrolní součet deskriptoru rámce, bloků a samotných dat a porovnávat ho s přítomným kontrolním součtem ve vstupních datech. Korektnost dat zajišťuje transportní vrstva síťového spojení. Odpadá tak nutnost implementovat hash funkci xxHash v dekompresoru, což povede k odlehčení objemu zdrojového kódu a ve výsledku k redukci potřebného množství zdrojů na FPGA čipu.

Ostatní funkce v popisu dekompresoru zůstanou zachované. Dekompresor musí být dostatečně obecný, aby dokázal zpracovat širokou škálu variant LZ4 archivů. Vstupem tak může být archiv pocházející z referenčního kompresoru nebo i z jiného kompresoru, který však nedodržuje dodatečná omezení. Dekompresor bude v tomto ohledu benevolentní a nebude některá pravidla striktně vynucovat. Z komprimovaného datového bloku může vzniknout větší objem výsledných dat než stanovuje maximální velikost bloku. Samotný blok však tuto stanovenou velikost přesáhnout nesmí. Dále datový blok může končit literály, stejně jak v implementaci autora LZ4, nebo match částí (LZ4 sekvence je pak ucelená). Dekompresor bude zvládat oba případy. Také není nutné dodržet pravidlo o výskytu poslední shody minimálně 12 bajtů před koncem plného datového bloku, které opět autor stanovil pro kompatibilitu se svou implementací. Dekompresor však bude předpokládat, že vstupní archiv je validní a tedy případné porušení zmiňovaných zásad je úmyslné, nikoliv vlivem chyb.

4.2 Kompresor

Programy poběží na uzlech s hustým síťovým provozem, a proto se očekává velké množství dat. Jednotlivé datové celky však musí dosahovat alespoň řádu mebibajtů, jinak by nasazení komprese nedávalo smysl. Maximální velikost datového bloku bude nastavena na maximální povolenou hodnotu, tedy na 4 MiB. Celý datový blok se bude uchovávat v bufferu v paměti, stejně jako v případě dekompresoru, neboť velikost bloku se nachazí před samotným blokem. Je tedy potřeba data v bloku ukládat, dokud nebude blok naplněn nebo ukončen a jeho velikost tak bude známa. I v kompresoru je nutné udržovat vyhledávací okno, neboť právě v něm se hledá shodná sekvence bajtů s aktuálním vstupem. Je tedy potřeba v paměti uložit vždy posledních 65 535 bajtů ze vstupních dat.

Problém představuje detekce konce vstupních dat, resp. konce datové transakce zmiňované u dekompresoru. V softwarové verzi postačuje kontrolovat konec souboru. V hardwarovém prostředí žádná abstrakce jako soubor k dispozici není, tudíž bude nutné vytvořit dodatečný signál zprostředkovávající tuto funkci a zahrnout ho do rozhraní. V případě dekompresoru není tato funkce tak kritická, neboť stačí, aby byl formát archivu korektní a k oddělení datových celků postačují jednotlivé rámce.

Jelikož budou vstupní data zpracovávána a zasílána sekvenčně, bude v deskriptoru datových rámců nastavena závislost datových bloků na předchozích blocích v rámci. Závislost bloků povede k lepšímu kompresnímu poměru. Nezávislost bloků by dávala smysl v prostředí, kde by bylo možné jednotlivé bloky zpracovávat paralelně. V tomto případě má

dekompresor výhradně sekvenční přístup k datům, takže v momentě začátku zpracovávání archivu nejsou všechna data dostupná, což zamezuje jejich paralelní dekompresi.

Data budou zasílána skrz síť, takže spolehlivý přenos bude zaručovat transportní vrstva (pravděpodobně s protokolem TCP), a proto i zde není potřeba nepovinné kontrolní součty počítat a zahrnovat. Velikost původních dat v deskriptoru nebude obsažena, neboť není potřebná a kompresor ji dopředu ani nezná. Hlavička rámce se nebude nijak měnit. Bude nastavena pevně na jednu konfiguraci, díky čemuž lze jediný povinný kontrolní součet v deskriptoru vypočítat předem a na pevno ho vložit do programu. Tímto se eliminuje nutnost implementovat hash funkci xxHash, stejně jako v případě dekompresoru.

4.2.1 Vyhledávací metoda

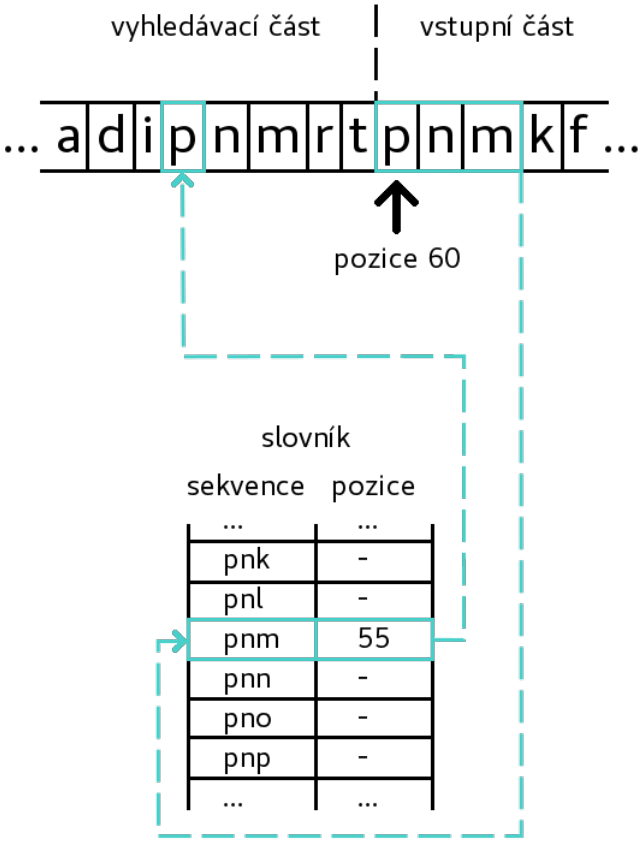
Algoritmus LZ4 nediktuje žádnou konkrétní metodu vyhledávání sekvencí. Je tedy možné vybrat vhodnou metodu podle požadovaných parametrů. Je velmi vhodné oddělit mechanismus vyhledávání sekvencí bajtů v okně od zbytku kódu programu. Různé vyhledávací metody je pak snadné porovnávat. V této sekci je uvedeno několik metod, ze kterých bude jedna vybrána a později implementována.

Přímý přístup

Jedná se o naivní a přímočarý přístup, který demonstruje dualitu paměťové a výpočetní náročnosti. V ideálním případě bychom chtěli shodnou sekvenci bajtů nalézt v konstantním čase. K tomu by však byla potřebná enormní paměťová kapacita, aby bylo možné uložit veškeré posloupnosti délek $4-65\,535$ bajtů ve vyhledávacím okně. Problém však není pouze v paměťové náročnosti, ale i ve stabilitě či trvanlivosti této struktury, neboť by bylo nutné ji neustále obnovovat. Je-li vyhledávací okno zaplněné, na jeho konec se přidá nový bajt a ze začátku je bajt vysunut. Tímto vznikne 65 535 nových posloupností, které je nutné zpracovat a přidělit jim příslušná metadata. Takto by byly shodné sekvence vždy nalezeny, ale za příliš velkou cenu.

Má-li být tento způsob použitelný, musíme snížit jeho paměťové i výpočetní nároky. Nebudou se ukládat posloupnosti všech délek, ale pouze posloupnosti pevné délky. Způsob vyhledávání je ilustrován na obrázku 4.1. Slovník tvoří tabulka, která obsahuje všechny kombinace hodnot dané délky. Každá položka tabulky je indexována samotnou posloupností a obsahuje informaci, zda se tato posloupnost ve vyhledávacím okně nachází a na které pozici, či nikoliv. Při hledání konkrétní sekvence mohou tedy nastat celkem 3 případy: posloupnost je v okně nalezena, posloupnost se v okně nevyskytuje a nebo se posloupnost v okně vyskytovala dříve, ale v momentě vyhledávání je již ve vyhledávacím okně přepsaná jinou posloupností. Z toho vyplývá, že je nutné kontrolovat platnost nalezených posloupností v tabulce. Dovolujeme-li určitou chybovost, nepřesnost či nepravdivost struktury, snižují se tak nároky na její údržbu.

Pokud bychom chtěli tuto metodu aplikovat, je nutné stanovit přijatelnou délku sekvencí bajtů. V případě LZ4 je minimální délka shody 4 bajty. Tabulka by tím pádem obsahovala 2^{32} položek. Každá položka musí obsahovat minimálně pozici ve vyhledávacím okně, pro kterou jsou potřebné 2 bajty. Celková velikost tabulky tedy vychází na 8 GiB minimálně. Informace, zda je položka v tabulce obsazená, není povinná, neboť v případě neshody aktuálního vstupu se sekvencí na pozici uvedené v dané položce tabulky dojde vyhledávání ke stejnému závěru. Zkrátíme-li délku posloupností na 3 bajty, výsledná dolní hranice velikosti tabulky bude 32 MiB, což už je použitelná varianta pro běžné počítače. Pokud bychom délky sekvencí opět o bajt zkrátili na 2 bajty, velikost tabulky dosáhne minimálních 128 KiB. Tuto



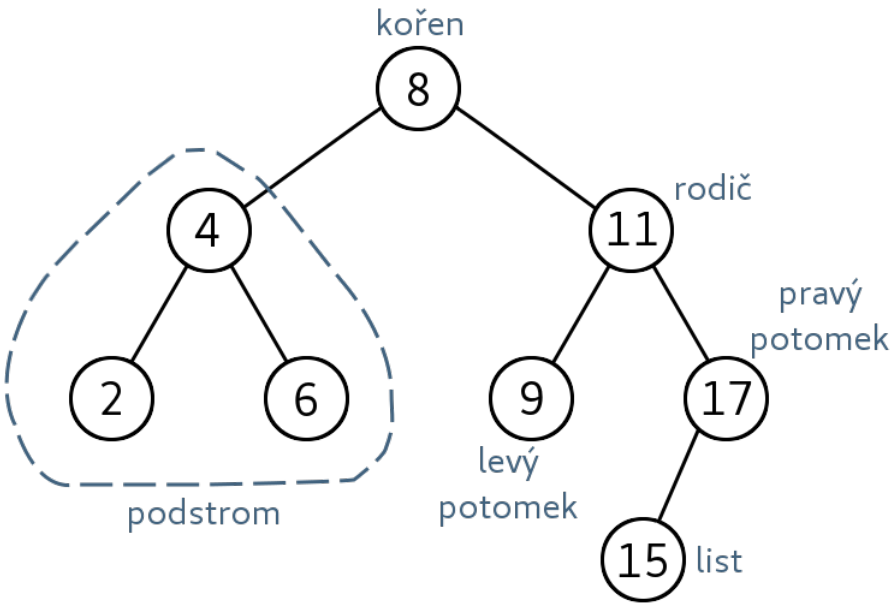
Obrázek 4.1: Ukázka vyhledávání pomocí tabulkového slovníku a přímého přístupu

variantu je možné použít i v prostředí FPGA. Posloupnost o 2 bajtech je však příliš krátká a ve vyhledávacím okně se může vyskytovat mnohokrát. V tabulce by se však nacházel pouze poslední výskyt těchto dvou bajtů, čímž by docházelo ke ztrátám mnoha použitelných shod. Proto se 3 bajty jeví jako ideální délka posloupností pro využití v LZ4. Po této uprávě ale zvýšíme konstantní časovou složitost vyhledávání. Sekvence na dané pozici v okně je nutné kontrolovat na platnost. Dále je potřeba v porovnávání bajtů pokračovat, neboť alespoň 4 bajty musí být shodné. Pokud shodné jsou, byla nalezena základní sekvence délky 4 bajty. Jelikož požadujeme shodné posloupnosti co nejdelší, porovnávání na úrovni bajtů bude pokračovat, dokud se odpovídající si bajty budou rovnat. Řešení tak nabývá lineární časové složitosti. Kvůli velikosti tabulky však nelze tento přístup v FPGA použít.

Binární vyhledávací strom

Binární vyhledávací strom, anglicky zkratkou BST (binary search tree), je uspořádaná datová struktura skládající se z uzlů. Každý uzel obsahuje minimálně hodnotu, neboli klíč, a dva ukazatele na další uzly – na levého a pravého potomka. Uzly nabývají specifických názvů splňují-li určité vlastnosti, což je znázorněno na obrázku 4.2. Uzel s hodnotou 11, který ukazuje na uzel s hodnotou 9, se nazývá rodič uzlu 9. Počáteční uzel, nazývaný

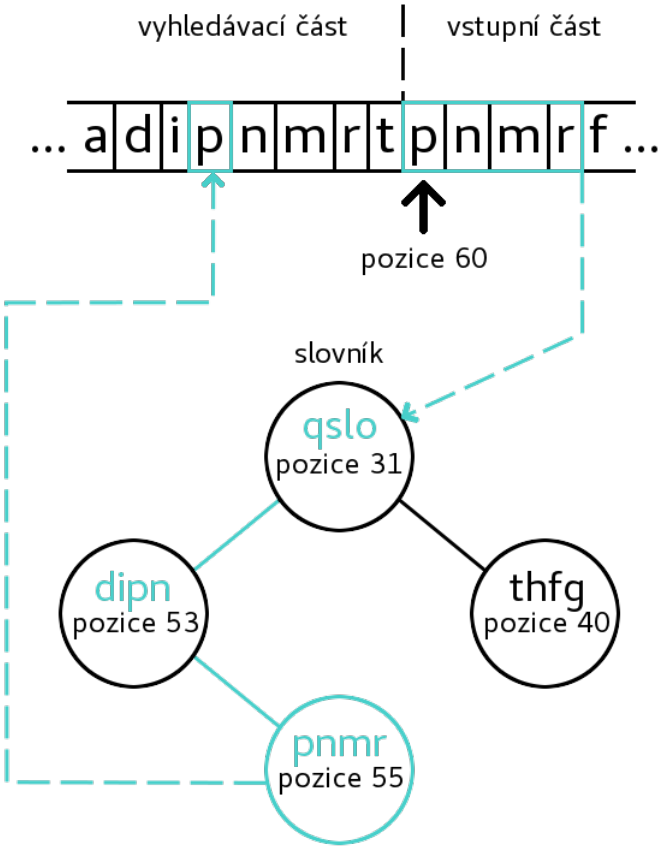
kořen, je jediný uzel bez rodiče. Uzly, které neukazují na žádného potomka, se označují jako listy stromu. Pokud strom obsahuje pouze kořen, je kořen zároveň i listem. Binární strom také obsahuje podstromy. Jedná se o část stromu skládající se z vybraného uzlu (kořene podstromu) a všech jeho potomků (a potomků jeho potomků, až k listům). Kořen podstromu, na rozdíl od kořene celého stromu, má rodiče. Je dobré podotknout, že strom samotný lze také označit jako podstrom. Jedná se o speciální případ.



Obrázek 4.2: Příklad BST a ruzných typů jeho uzlů

Označení potomků jako levý a pravý má své odůvodnění. BST je uspořádaný podle hodnot klíčů. Při vkládání nového uzlu se porovnává klíč nového uzlu s ostatními. Začíná se od kořene. Je-li vkládaný uzel menší než kořen (ve skutečnosti jsou porovnávány jejich klíče, ale pro zjednodušení se dále bude pracovat s celými uzly), pokračuje se v porovnávání s levým potomkem kořene. V případě, že je vkládaný uzel větší, porovnává se dále s pravým potomkem kořene. Takto se pokračuje, dokud porovnávaný uzel ve stromě nemá dalšího potřebného potomka. V této situaci je uzel vložen na pozici chybějícího potomka. Ze způsobu vkládání uzlů vyplývá, že u libovolného kořene platí, že všechny uzly v jeho levém podstromu jsou menší než kořen a veškeré uzly v pravém podstromu jsou vetší než kořen, jak je možné vidět na obrázku 4.2.

Díky uspořádání stromu lze efektivně vyhledávat uzly s určitou hodnotou. Časová složitost této operace je v průměru rovna $O(\log_2(n))$, kde n je počet uzlů ve stromu [7]. Princip BST lze aplikovat v kontextu algoritmu LZ4, je ale nutné provést rozšíření. Uzel bude obsahovat ukazatele na potomky, klíč, který bude roven hledané sekvenci bajtů, a navíc položku s pozicí určující, kde tato sekvence ve vyhledavacím okně začíná. Strom bude uspořádaný podle hodnoty sekvencí bajtů. Při vyhledávání mohou opět nastat 3 případy: uzel s hledanou sekvencí se ve stromu nenachází a sekvence je vložena jako nový listový uzel do stromu, uzel se sekvencí je úspěšně nalezen a pozice v uzlu odpovídá skutečnému výskytu sekvence ve vyhledávacím okně nebo je uzel nalezen, ale pozice v něm uložená odkazuje na již neplatné umístění hledané posloupnosti ve vyhledávacím okně. Vyhledání je demonstrováno na obrázku 4.3.



Obrázek 4.3: Příklad úspešného vyhledání sekvence v BST

Nyní je nutné se zaměřit na parametry této varianty a její vhodnost pro prostředí FPGA. Výhodou BST jako slovníku oproti popisovanému přímému přístupu je menší paměťová náročnost, neboť není nutné mít předem alokované položky pro všechny kombinace posloupností bajtů. Záznamy o sekvencích jsou ve formě uzlů dynamicky vytvářeny tak, jak s nimi kompresor postupně přichází do styku, čímž se eliminují nevyužité položky a zefektivní se práce s pamětí.

Je však zřejmé, že strom nemůže růst do libovolné velikosti a celkový počet uzlů bude nutno limitovat. Aby bylo možné limit alespoň přibližně určit, je zapotřebí stanovit délku sekvencí v uzlech. Posloupnosti bajtů je možné porovnávat několika způsoby. Jedna z aplikovatelných metod je lexikografické porovnání, tedy porovnávání bajt po bajtu, díky čemuž lze používat variabilní délky posloupností. Jelikož uzel v takovém případě nemá pevně stanovenou velikost, je nutné evidovat množství obsazené paměti. Při vkládání nového uzlu by se kontrolovalo, zda je dostupná potřebná paměťová kapacita pro uložení posloupnosti v uzlu. Další způsob porovnání sekvencí je pomocí konkatenace bajtů sekvence do jedné hodnoty. Použití této metody není výhodné v kombinaci s variabilní délkou posloupnosti, neboť je potřeba konkatenovanou hodnotu reprezentovat číselným datovým typem. Proto je vhodné posloupnosti v uzlech omezit na pevnou délku, nejlépe na 4 bajty, jak je zobrazeno na obrázku 4.3. Jedná se již o sekvenci platné délky a zkonkatenovanou hodnotu je možné uložit

do 32bitové proměnné. Klíč obsazuje 4 bajty, pozice ve vyhledávacím okně 2 bajty. Po odečtení velikosti vyhledávacího okna a maximální velikosti bloku od dostupné kapacity paměti na FPGA čipu získáme zbývající použitelnou kapacitu paměti, neboť obě datové struktury musí kompresor obsahovat. Pomocí zbývající velikosti paměti lze rovnicí (4.1) vypočítat velikost ukazatelů uzlu na potomky, kde x po zaokrouhlení nahoru určuje počet bajtů na jeden ukazatel. Rovnice nelze vypočítat analyticky, tudíž je potřeba aplikovat numerické metody. Pro tuto situaci přibližný výsledek nečiní problém. Každý ukazatel na potomka potřebuje 3 bajty. Ve skutečnosti stačí k adresování všech uzlů pouze 17 bitů. V klasickém softwaru není možné vytvářet proměnné, které nejsou zarovnané na bajt. V hardwaru a v FPGA však není registr o 17 bitech problém. Dohromady je tedy potřeba 82 bitů na uzel. Pokud by se použila veškerá zbývající paměť, celkem by bylo možné vytvořit kolem 117 485 uzlů.

$$\begin{aligned} \frac{\text{zbývající paměť v bajtech}}{6+2x} &\leq 2^{8x} \\ \frac{1204225}{6+2x} &\leq 2^{8x} \\ x &\geq 2,10594 \end{aligned} \tag{4.1}$$

Varianta slovníku s BST má však své problémy a negativa. Strom musí zachovávat svou uspořádanou strukturu. Uspořádávání probíhá při každém vkládání nového uzlu do stromu, neboť je vkládán na příslušnou pozici podle pravidel BST. V podstatě se jedná o operaci vyhledání doplněnou o navázání uzlu na slepý ukazatel listu stromu, což opět nabývá logaritmické časové složitosti. Jelikož je uzel nejdříve vyhledán a pokud není nalezen, tak je vložen, lze si poslední navštívený list uložit a urychlit tak případné vkládání.

Počet uzlů ve stromu je omezen výše zmiňovanou hodnotou. Po dosažení maximální hodnoty je možné aplikovat jedno ze 2 možných chování: žádné nové uzly se již nebudou vytvářet nebo existující uzly budou mazány a nové místo nich vkládány. První volba zaplněný strom transformuje na statický slovník, což má dopad na jeho flexibilitu a adaptabilitu na vstupní data. Tím se zhoršuje dosažitelný kompresní poměr metody. Druhá volba zachovává flexibilitu slovníku za cenu dodatečné režie.

Otázkou také zůstává způsob výběru uzlu k odstranění, neboli tzv. oběti. Uzel může být vybrán náhodně. Odstraňovaný uzel pak může být listem, uzlem s potomky nebo i kořenem. List lze odstranit jednoduše. V případě kořene stačí prohlásit jeho levého nebo pravého potomka za nový kořen. U uzlu s potomky a rodičem je nutné správně napojit ukazatele podle pravidel stromu. Nelze jej odstranit okamžitě, neboť je zapotřebí získat jeho rodiče, což je možné pouze průchodem stromu. Aby bylo možné takový uzel odstranit v konstantním čase, musel by každý uzel obsahovat zpětný ukazatel, který by odkazoval na rodiče uzlu. Přidání další položky, a tedy zvětšení objemu uzlu, by však dále limitovalo maximální počet uzlů.

S vkládáním a mazáním uzlů také souvisí údržba stromu. V nejhorším případě může strom zdegradovat na lineárně vázaný seznam a logaritmická časová složitost vyhledávání by se zvýšila na lineární. Proto je nutné strom vyvažovat přesouváním jeho podstromů tak, aby byla jeho struktura v souladu s pravidly BST zachována.

I když lze sekvenci délky 4 bajty vyhledat v logaritmickém čase, je opět nutné sekvenci ověřit. V případě, že je neplatná, je daný uzel smazán nebo ponechán ve stromě podle předchozí zvolené varianty. Pokud sekvence odpovídá, je nutné provádět kontrolu následujících bajtů na shodu lineárně, tudíž výsledná časová složitost vyhledání je jako v případě přímého

přístupu lineární. Jelikož hlavní požadavek je rychlost, režie spojená s údržbou struktury nečiní z BST slovníku příliš vhodné řešení.

Hash tabulka

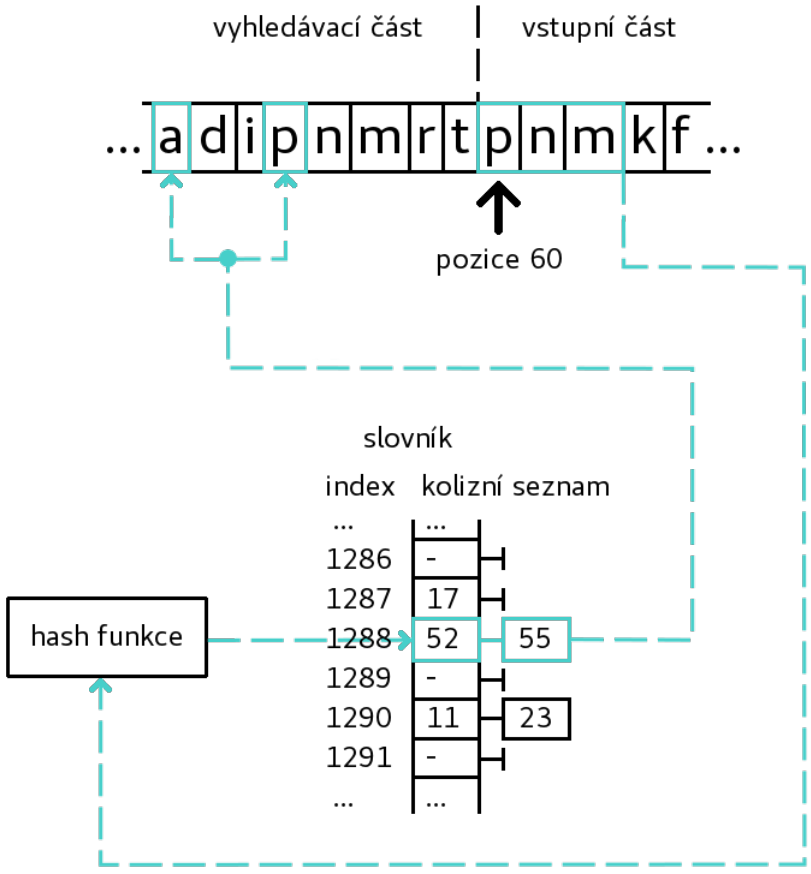
Hash tabulka je druh asociativní datové struktury skládající se ze dvou částí [7]. První komponenta se nazývá hash funkce. Jedná se o funkci, která přijímá libovolný objem vstupních dat a na výstup produkuje data konstantní délky. Výstup funkce určitým způsobem identifikuje vstupní data. Jelikož vstup může být delší, dochází ke ztrátě informace. Zobrazení tedy není bijektivní, a tudíž 2 nebo více odlišných vstupů může dostat přidělený stejný výstup. V kontextu hash funkcí se tento jev nazývá kolize. Existuje třída kryptografických hash funkcí, které musí splňovat dodatečné požadavky, mezi které patří bezkoliznost, resp. vytvoření kolize je výpočetně nezvládnutelné. Jelikož má výstup funkce konstantní délku, lze ho interpretovat jako index do úložné datové struktury. Tím se dostáváme k druhé složce hash tabulky a to k samotné tabulce. Celá struktura je zjednodušeně řečeno pole, které používá vkládané položky pro generování jejich příslušných indexů. Srovnáme-li hash tabulku s dříve popisovanou metodou přímého přístupu, najdeme několik zásadních odlišností. Přímý přístup, jak název napovídá, používá přímo vstupní data pro indexování pole. Hash tabulka používá hash funkci pro derivaci indexu do pole ze vstupních dat. Hash funkci lze tedy chápat jako zprostředkovatele přístupu k tabulce. Tabulka u přímého přístupu má vyhrazenou položku pro každý vstupní prvek, kdežto u hash tabulky běžně nastává situace, kdy má více prvků obsazovat jednu položku. Proto se používá pojem mapování na košík (anglicky bucket). Jedná se o množinu kolizních prvků pro daný index získaný z hash funkce.

Hash tabulku lze jednoduše aplikovat na problematiku LZ4. Vstupní data hash funkce představují posloupnosti bajtů, ze kterých získáme data interpretovatelná jako číslo na určitý počet bajtů. Délka závisí na konkrétní zvolené hash funkci, což přímo ovlivňuje velikost hash tabulky. Při rozebírání přímého přístupu jsme zjistili, že tabulka s pozicemi začátků posloupností ve vyhledávacím okně může obsahovat 2^{16} položek. Hledáme tedy funkci, která zvládá zpracovat řetězec bajtů libovolné délky a jejím výstupem jsou vždy 2 bajty. Přidáme-li požadavek na rychlost v hardware, CRC-CCITT (Kermit) [4] se jeví jako vhodný kandidát.

Nyní je nutné zvolit datovou strukturu pro množiny kolizních prvků. Struktura může být naprosto libovolná. Je možné do tabulky připojit např. binární vyhledávací stromy, čímž lze dosáhnout dvoustupňového vyhledávání. Prostředí FPGA je značně paměťově omezené, a proto je snaha volit jednodušší konstrukce. Nevýhody BST již byly v této sekci zmíněny. Bylo by potřeba udržovat až 2^{16} stromů. Dále by se musel hlídat počet uzlů ve stromech. Každý strom by měl buď přidělený maximální počet uzlů nebo by se uzly počítaly globálně napříč všemy stromy. Proto je vhodnější použít klasický kolizní seznam, což je lineárně vázaný seznam prvků.

Seznam může být statické nebo dynamické délky. Dynamická délka seznamu s sebou nese stejnou paměťovou režii jako v případě BST. S ohledem na kapacitu paměti a výkon FPGA je rozumnější seznam staticky omezit na konzervativní délku 2 položky. Později může být seznam případně zvětšen. Jelikož je délka kolizního seznamu takto omezena, je vhodné odpovídajícím způsobem také limitovat délku vstupních dat do hash funkce na 3 bajty. Prostor 24bitových hodnot je tak mapován na prostor 16bitových hodnot, čímž se oproti libovolně dlouhému vstupu výrazně redukuje maximální počet kolizních prvků v jednom košíku. Ostatně hash 3bajtové sekvence se používá i v rychlém kompresním režimu refe-

renční implementace autora (mód s přepínačem -1). V kolizním seznamu se budou nacházet indexy začátků 3bajtových posloupností ve vyhledávacím okně s odpovídajícím hash indexem, podobně jako v případě přímého přístupu. Přidáme-li ke každé položce seznamů 1bitovou hodnotu pro určení, zda je daná položka obsazena, jeden kolizní seznam (tedy záznam v hash tabulce) potřebuje 2 16bitové indexy do vyhledávacího okna a 2 1bitové příznaky, celkem 34 bitů. Celá hash tabulka tedy spotřebuje kolem 272 KiB.



Obrázek 4.4: Příklad vyhledání sekvence v hash tabulce

Struktura hash tabulky je velice nenáročná na údržbu, neboť nevyžaduje žádné specifické uspořádání jako v případě binárního vyhledávacího stromu. Pro nalezený index v tabulce je nutné zkontrolovat, zda je sekvence na daném indexu ve vyhledávacím okně přítomná. Jelikož je délka vstupní sekvence do hash funkce rovna 3 bajtům, nejedná se o plnohodnotnou použitelnou sekvenci, neboť nesplňuje minimální délku 4 bajtů. Proto je nutné dodatečně porovnat následující bajt, aby bylo možné vyhledání prohlásit za úspěšné, jinak se pokračuje procházením kolizního seznamu. Vyhledání posloupnosti bajtů v tabulce je ilustrováno na obrázku 4.4. Pokud posloupnost není nalezena, aktuálně zpracovávaný index je vložen do kolizního seznamu. V případě plného seznamu je starší záznam vkládaným indexem nahrazen. Stejně jako v případě uváděné varianty BST, i zde je nutné porovnávat bajt po bajtu k získání co nejdelší sekvence. Hash tabulka tedy představuje kompromis mezi rychlostí, spolehlivostí vyhledání a složitostí datové struktury, a proto bude použita pro implementaci

slovníku v LZ4. Navíc lze nastavovat řadu parametrů: samotnou hash funkci, délku kolizních seznamů nebo chování v určitých situacích, což umožňuje vytvářet konfigurace s různými vlastnostmi, ze kterých lze na základě srovnání vybrat tu nejlepší z nich. Distribuce hodnot hash funkcí do košíku je jednou z fundamentálních vlastností. Je snahou volit funkce, které rozdělují prostor vstupních hodnot rovnoměrně, což vede k efektivnímu využití všech položek tabulky a jejich kolizních struktur. Tím se zvýší šance nalezení sekvencí v případě omezené velikost košíku.

4.3 Hardwarové rozhraní

Prostředí FPGA čipu nemá žádné ponětí o konceptu souborů, a proto je nutné vytvořit nové nebo zvolit již existující vhodné rozhraní pro předávání dat. Toto rozhraní musí být schopno přenášet libovolný objem dat a informovat o konci dat a začátku nových.

4.3.1 FLU

FLU je vysokorychlostní, paketově orientované rozhraní, které se používá v rámci FPGA čipů pro přenos síťových rámců či obecných dat [5]. Tabulka 4.1 ukazuje jednotlivé signály rozhraní. Na tabulku navazuje popis těchto signálů.

Název signálu	**Šířka [b]**	**Směr**
CLK	1	–
DATA	$DATA\_WIDTH$	zdroj → cíl
SOP	1	zdroj → cíl
SOP_POS	$\langle 1, \log_2(\frac{DATA\_WIDTH}{8}) \rangle$	zdroj → cíl
EOP	1	zdroj → cíl
EOP_POS	$\log_2(\frac{DATA\_WIDTH}{8})$	zdroj → cíl
SRC_RDY	1	zdroj → cíl
DST_RDY	1	cíl → zdroj

Tabulka 4.1: Signály tvořící rozhraní FLU

- *CLK (Clock):* synchronizační signál hodin.
- *DATA:* sběrnice, pomocí které se přenášejí kusy samotných data. Data na sběrnici jsou platná pouze pokud jsou oba signály SRC_RDY a DST_RDY nastaveny na hodnotu 1. Šířka sběrnice $DATA\_WIDTH$ může být libovolná, obvykle se však jedná o mocninu 2.
- *SOP (Start Of Packet):* indikuje začátek nového datového rámce v přenášeném datovém slově, pokud je tento signál nastaven na 1. Hodnota signálu je platná pouze pokud jsou oba signály SRC_RDY a DST_RDY nastaveny na 1.
- *SOP_POS (Start Of Packet Position):* určuje pozici začátku nového rámce v datovém slově na sběrnici. Šířka tohoto signálu je dána parametrem SOP_POS_WIDTH, který je závislý na konkrétní cílové platformě. Pozice začátku rámce se vypočítá vztahem (4.2). Tento signál je platný pouze pokud jsou platné signály SOP, SRC_RDY a DST_RDY.

$$\text{pozice začátku rámce} = SOP\_POS \cdot \frac{DATA\_WIDTH}{2^{SOP\_POS\_WIDTH}} \tag{4.2}$$

- *EOP (End Of Packet):* indikuje konec datového rámce v přenášeném datovém slově, pokud je nastaven na hodnotu 1. Signál je platný pouze pokud jsou oba signály SRC_RDY a DST_RDY nastaveny na 1.
- *EOP_POS (End Of Packet Position):* určuje pozici konce přenášeného rámce v datovém slově na sběrnici. Signál musí být tak široký, aby bylo možné adresovat libovolný bajt na datové sběrnici, který je posledním v daném datovém rámci. Tento signál je platný pouze pokud jsou platné signály EOP, SRC_RDY a DST_RDY.
- *SRC_RDY (Source Ready):* udává, zda je odesílatel připraven odeslat data. Data jsou úspěšně odeslána pouze pokud je spolu se signálem DST_RDY nastaven na 1, což může nastat ve stejném nebo dalším hodinovém taktu.
- *DST_RDY (Destination Ready):* udává, zda je příjemce připraven data přijmout. Data mohou být přijata pouze pokud je připraven i odesílatel, tedy pokud jsou oba signály SRC_RDY a DST_RDY nastaveny na 1.

Toto rozhraní splňuje výše uvedené požadavky. Umožňuje načítat více bajtů zároveň, které by bylo možné zpracovávat paralelně. Jeden z mála problémů by však představovala nutnost implementovat vícenásobný přístup do slovníku a zajištění jeho konzistentního stavu. Už jen režie spojená s obsluhou slovníku by řešení značně zkomplikovala.

4.3.2 AC Channel

Knihovna Algorithmic C, dostupná v nástroji Catapult, poskytuje šablonovou C++ třídu `ac_channel` [8]. Jedná se o abstraktní datový typ fronta, jehož prvek může být libovolný datový typ. Jelikož je fronta typu FIFO (First In, First Out), jsou prvky v ní uchované čteny ze začátku fronty ve stejném pořadí, jak byly na její konec zapisovány. Při manipulaci s daty uvnitř `ac_channel` (čtení, zápis) dochází k tzv. handshake, což je způsob synchronizace čtecí a zápisové jednotky. Pokud se čtecí jednotka snaží načíst prvek z fronty, ale fronta je prázdná, čtení se stane blokujícím a tento VHDL proces bude pozastaven, dokud ve frontě nebude alespoň jeden prvek. Poté je možné čtení dokončit a pokračovat ve výpočtu. Podobně v případě zápisu je-li fronta plná. Zápisový VHDL proces je pozastaven, dokud není ve frontě volné místo pro zapisovaný prvek. Tímto je zajištěna ochrana proti podtečení a přetečení vnitřního bufferu fronty.

Aby rozhraní vytvořené pomocí `ac_channel` splňovalo potřebné požadavky, datový typ prvku bude struktura `Interface_t` skládající se z komponent popsaných v tabulce 4.2. Data se budou zpracovávat sériově po jednotlivých bajtech. Při startu se automaticky předpokládá začátek nových dat a díky dříve popisovanému chování fronty bude kompresor nebo dekompresor čekat na příchod dat. Pokud je položka IsLastByte nastaven na 1, uchovaný bajt v položce Data je poslední v právě zpracováváném datovém celku a následující položka ve frontě je již součástí nového datového celku. Začátek dat je tedy rozpoznán implicitně na základě startu běhu programu nebo indikovaného konce dat.

Název položky	**Šířka [b]**
Data	8
IsLastByte	1

Tabulka 4.2: Komponenty struktury `Interface_t`

Kapitola 5

Implementace

Vzniklou verzi pro transformaci ze softwarové není nutné striktně nazývat hardwarovou. Jedná se totiž o jistý mezistupeň či hybridní implementaci, neboť používaný implementační programovací jazyk je stále s jistými restrikcemi jazyk C/C++. Avšak spolu s knihovnou Algorithmic C a příslušnými nástroji v prostředí Catapult je možné tuto verzi do jisté míry automatizovaně přeložit do VHDL kódu a použít tak na platformě FPGA. Lze tedy hardwarovou verzi programu označit také jako syntetizovatelnou. Lze ji také přeložit pro použití na klasických procesorech, což je velice výhodné pro účely verifikace a testování.

5.1 Softwarová verze

Kompresor i dekompresor budou v čistě softwarové podobě napsané v jazyce C. Nabízí se varianta implementace v C++, neboť je jazyku v prostředí Catapult bližší. Z důvodu srovnání s referenční implementací autora byl však zvolen jazyk C. Objektově orientovaný přístup by kvůli navazující transformaci na hardwarovou implementaci pravděpodobně nebyl využit. Navíc je jazyk C z vysokoúrovňových procedurálních jazyků z hlediska abstrakce nejblíže fyzickému hardwaru, a tudíž mají programy v něm napsané největší předpoklady pro nejrychlejší běh.

Jelikož mají být výsledné programy jednoúčelově vestavěné v FPGA čipu, bude veškerá uživatelská interakce odstraněna i v softwarové verzi. Prostředí streamového zpracování na síťové kartě bude simulováno přesměrováváním dat. Kompresor a dekompresor budou znát pouze 2 datové cesty, podobně jako v případě hardwarové (hybridní) verze – standardní vstup a výstup.

5.1.1 Dekompresor

Ve specifikaci LZ4 algoritmu je stanoveno, že alespoň jedna konfigurace parametrů obsažených v deskriptoru musí být pro dekompresor akceptovatelná, ne nutně všechny. Je však nutné zhodnotit množinu možných vstupních archivů a požadavky na jejich zpracování. Z hlediska implementace je nejjednodušší se omezit na určitou podmnožinu konfigurací deskriptoru. Není však přesně určeno, z jakého zdroje budou archivy pocházet, jaký kompresor bude použit a zda splňuje omezení stanovená referenční implementací či nikoliv. Proto je vhodné zvolit opačnou strategii – záměrně nestanovovat žádné předpoklady o tvaru LZ4 archivu a snažit se být kompatibilní s co nejširší škálou konfigurací.

Slovník a datové struktury

Ústřední datovou strukturou je zde `Lz4Context`. Obsahuje komponenty, které určují aktuální stav dekomprese. Ukazatel `BlockHolder` bude odkazovat na aktuálně zpracovávaný datový blok, který bude celý uložený v paměti. Pole `Window` představuje vyhledávací okno, což je také slovník pro dekompresor. Na jeho konec jsou vkládána dekomprimovaná data. Velikost slovníku je přesně 65535 bajtů, což splňuje požadavek hodnoty maximálního offsetu. Položka `DataSize` určuje aktuální počet bajtů nacházejících se ve slovníku. Je tak možné kontrolovat chybné offsety (větší offset než je reálné množství dat ve slovníku). Poslední část struktury tvoří soustava 3 ukazatelů odkazujících vyhledávací okno `Window`. `WindowPosition_Initial` ukazuje na začátek pole, `WindowPosition_Current` uchovává pozici za posledním bajtem v poli a ukazatel `WindowPosition_Maximum` odkazuje pozici za posledním prvek mimo kapacitu pole – slouží jako zarážka. Tyto 3 ukazatele budou používány pro manipulaci s vyhledávacím oknem (kopírování literálů, vykonávání operace match, přeposílání dekomprimovaných dat na výstup). Ostatní důležité proměnné se vyskytují samostatně s deklarací v místech, kde jsou potřebné. Primární účel sdružení proměnných do struktury je častá práce s nimi a redukce počtu parametrů při vytváření procedur a funkcí, neboť ve většině případů vykonávání důležitějších operací se pracuje alespoň s jednou komponentou v `Lz4Context`.

Inicializace a zpracování metadat

Nejdříve probíhá inicializace struktury `Lz4Context`. Nastavuje se trojice ukazatelů do vyhledávacího okna na své patřičné pozice a nuluje se počet bajtů v okně `DataSize`. Samotné vyhledávací okno `Window` není třeba žádným způsobem inicializovat, neboť bajty se nejdříve do pole budou zapisovat a až poté číst. Tím pádem budou čtena vždy správná data. Dále se již začínají načítat vstupní data ze standardního vstupu, který reprezentuje vstupní datový kanál ve výsledné FPGA architektuře. Předpokládá se, že poskytnutá vstupní data jsou součástí LZ4 archivu, a proto jsou některé typické kontroly vstupních dat vynechány, např. ošetření předčasného konce vstupních dat.

Po inicializaci je možné začít se vstupními daty. Načtou se první čtyři bajty, které správně obsahují magickou konstantu jednoho z typů rámců. Drtivá většina následujících konstrukcí se nachází uvnitř `while` smyčky, která se bude opakovaně provádět, dokud nebudou veškerá vstupní data vyčerpána. Konec dat lze v tomto případě jednoduše určit funkcí `feof`, která informuje, zda se program nachází na konci souboru, jenž je zadaný jako parametr funkce. Jelikož je standardní vstup abstrahován také jako soubor, lze tuto funkci využít. Kód uvnitř smyčky je až na jedno volání funkce celý zapouzdřen v jednom větvení. Celé zpracování LZ4 archivu lze rozdělit na 2 velké celky – zpracování LZ4 (datového) a uživatelského rámce. Archiv se skládá ze za sebou jdoucích rámců. Rámec může jednoho ze zmiňovaných typů. První větev řeší jednodušší situaci s uživatelským rámcem, ve které je načtena velikost rámce a celý rámec je přeskočen jeho pouhým načítáním, jak bylo popsáno v návrhu v sekci 4.1. Druhá větev řeší samotnou dekompresi, tedy datový rámec. Vždy po zpracování jednoho z rámců ve větvení se program pokusí ze vstupu načíst další magickou konstantu případného následujícího rámce. Načtením se ukončí `while` smyčka a vyhodnotí se podmínka s kontrolou konce vstupu funkcí `feof`. Pokud by se na vstupu žádný další rámec nevyskytoval, do smyčky by se nevstoupilo a zpracovávání by se tak ukončilo.

Druhá větev je jádrem dekompresního programu. Nejprve se zkontroluje, zda načtená magická konstanta identifikuje LZ4 rámec. Není-li tomu tak, program končí s chybovým kódem. Veškeré návratové kódy programu jsou logicky sdruženy do jednoho výčtového typu

`Lz4ReturnValues_t`. Je-li magická konstanta správná, započne získávání parametrů rámce z deskriptoru. Načte se jeho 1 bajt ze vstupu. Zkontroluje se verze rámce a nulové hodnoty rezervovaných bitů, načte se nastavení volby závislosti jednotlivých datových bloků, nastavení přítomnosti celkové velikosti původních dat a nastavení přítomností kontrolních součtů za datovými bloky a na konci celého rámce. Pokud některý rezervovaný bity nulový není, je program s příslušným návratovým kódem ukončen. Druhá část deskriptoru se nachází v dalším načteném bajtu, ve kterém se opět zkontroluje nulovost rezervovaných bitů. Především se ale z něho získá maximální extrahovatelná velikost dat z datových bloků v tomto rámci. Jelikož jsou ostatní bity v bajtu nulové, stačí na celý bajt aplikovat operaci bitového posuvu o 4 pozice doprava, čímž se jednoduše extrahuje bitový identifikátor, podle kterého je možné z tabulky 3.1 určit maximální objem původních dat v datových blocích. Je-li identifikátor menší než 4, nejedná se o platnou hodnotu a zpracovávání archivu je s návratovým kódem situace ukončeno. Odpovídající velikost v bajtech lze vypočítat podle vztahu (5.1). V programu je umocňování a násobení dvěma z rovnice (5.1) nahrazeno operacemi bitového posuvu doleva. Posunutí všech bitů v dané proměnné doleva je ekvivalentní operaci jejího vynásobení dvěma, avšak je mnohem rychlejší. Pokud je z deskriptoru zjištěna přítomnost původní velikosti dat, je nyní těchto 8 bajtů přeskočeno. Posléze stačí přeskočit 1bajtový kontrolní součet rámcového deskriptoru, a tím jsou veškerá metadata rámce zpracovaná.

$$\text{maximální objem dat} = 65536 \cdot 2^{2(\text{bitový identifikátor} - 4)} \quad [\text{B}] \tag{5.1}$$

Datové bloky

Po vyřízení hlavičky LZ4 rámce se na vstupu nachází předem neznámý počet datových bloků. Každý blok začíná 4bajtovým popisem své velikosti. Z deskriptoru byl dekódován maximální objem dat, který je možné z bloků jakýmkoliv způsobem extrahovat. Podle tohoto parametru se dynamicky alokuje pole `BlockHolder` ve struktuře `Lz4Context`, které bude schopno uchovat celý datový blok. Se vstupním kanálem se tedy bude pracovat nárazově a načítat se budou velké datové celky, což je z hlediska klasických počítačových systémů požadované a optimální chování. Po alokaci se načte velikost bloku. Nyní může nastat řada situací. Nejdříve se testuje, zda není velikost bloku nulová a samotný blok tedy prázdný. Prázdný blok signalizuje konec posloupnosti bloků, čímž končí i LZ4 rámec. Jedná se o koncovou značku znázorněnou na obrázku 3.2. Specifikace LZ4 algoritmu však neuvádí, zda musí být tento blok komprimovaného nebo nekomprimovaného typu. Proto je nutné počítat s oběma případy. Kontrola se provede snadno s pomocí bitového posuvu o jednu pozici doleva. Případný jedničkový bit na nejvyšší bitové pozici (MSB) je vysunut a hodnoty ostatních bitů, nulových či nenulových, jsou zachovány. Ze strany nejmenší bitové pozice (LSB) je automaticky při bitovém posuvu nasunut bit s hodnotou 0. Na pozici bitů momentálně nezáleží, postačuje informace o (ne)nulovosti celé proměnné.

Je-li blok opravdu prázdný, zpracovávání aktuálního rámce končí. Podle načteného příznaku z deskriptoru v `IsContentChecksumPresent` se případně přeskočí 4bajtový kontrolní součet, vynuluje se čítač bajtů ve vyhledávacím okně a veškerá data z okna jsou vysunuta na výstup, resetuje se pozice v okně na začátek a program se pokusí načíst další rámec. V opačném případě se pomocí bitové masky a operace bitového součinu k odstínění příznaku typu bloku zkontroluje, zda blok neporušuje v deskriptoru stanovenou maximální velikost dat v bloku. Tato kontrola dává především smysl v případě nekomprimovaných bloků, neboť data z nich nijak dál neexpandují. Komprimované bloky budou správně vždy menší než maximální povolená velikost právě díky aplikované kompresi. Kontrola tedy může upozor-

nit na bloky nesplňující specifikaci a při nalezení takového bloku je program s chybovým návratovým kódem ukončen. Nyní se program opět rozděluje do dvou větví – zpracování komprimovaného a nekomprimovaného bloku. Typ bloku se zjistí opět pomocí operace bitového součinu velikosti bloku a bitové masky.

Zpracování nekomprimovaného bloku

Celé zpracování nekomprimovaného bloku se odehrává v proceduře `copyFromBlock2Window`. Tato procedura se v programu používá pro kopírování dat z nekomprimovaných bloků a kopírování literálů do vyhledávacího okna. Veškerá extrahovaná data musí být vložena na konec okna, aby bylo možné match sekvence korektně expandovat. Mohou zde nastat 2 situace. Všechna data z nekomprimovaného bloku je možné umístit do okna, tedy velikost bloku je menší nebo rovna volné kapacitě v okně. V takovém případě stačí pomocí funkce `memcpy` zkopírovat celý obsah bloku do okna, posunout ukazatel `WindowPosition_Current` a zvýšit čítač `DataSize` o velikost bloku. Druhý případ představuje situaci, kdy okno nedisponuje dostatečnou volnou kapacitou pro data z bloku. Je tedy nutné patřičný počet bajtů ze začátku okna vysunout a nová data na konec okna nasunout. Pokud jsou data vetší než samotné okno, je možné bajty z okna vysunout v jednom kroku a zkopírovat z bloku data o velikosti okna. Pokud je ale blok menší než okno, je nutné provést dodatečné kroky. Nejprve jsou data vysunuta ze začátku okna, poté je zbylý obsah přesunut na začátek okna a až poté je možné nové bajty z bloku do okna zkopírovat. Právě při vysouvání dat z vyhledávacího okna nastává moment, kdy jsou dekomprimovaná data posílána na standardní výstup funkcí `fwrite`.

Zpracování komprimovaného bloku

Jelikož je velikost bloku uložena v proměnné `BlockSize`, lze celý blok načíst do alokovaného pole `BlockHolder` naráz. Se vstupním kanálem tak není potřeba do načítání dalšího bloku pracovat. Proměnná `BlockHolder_Index` se bude používat pro pohyb v bloku a jeho postupné zpracování. Komprimovaný datový blok se skládá z LZ4 sekvencí. Tyto sekvence se budou stejným způsobem opakovaně zpracovávat ve smyčce `while`, dokud je `BlockHolder_Index` menší než velikost bloku, tedy dokud není celý blok zpracován. LZ4 sekvence obsahují literálovou část a match část. Na začátku sekvence se vždy nachází token. Ke zjištění počtu literálů je potřeba extrahovat informaci z jeho horních 4 bitů. Do proměnné `LiteralSize`, která bude obsahovat výslednou velikost literálů v aktuální sekvenci, se uloží bitově posunutá hodnota tokenu o 4 bity doprava, čímž se jednoduše získá požadovaná hodnota horních 4 bitů. Pokud je již nyní počet literálů nulový, přechází se ke zpracovávání match části. V případě nenulové hodnoty `LiteralSize` se ověří, je-li hodnota rovna 15. Pokud ano, znamená to, že se za tokenem nachází dodatečné bajty popisující počet literálů v sekvenci. Přičtení těchto bajtů zaručí velmi krátký `while` cyklus, který bude hodnoty bajtů přičítat k `LiteralSize`, dokud není hodnota bajtu menší než 255, jak je určeno specifikací. Po zjištění definitivního počtu literálů se opět zavolá procedura `copyFromBlock2Window`, která literály vloží do vyhledávacího okna. V tomto případě se využije parametr procedury `Index`, který slouží k rozeznání případu kopírování nekomprimovaného bloku od literálů (používá se v obou situacích) a zároveň v případě literálů určuje pozici v načteném bloku, odkud se mají literály kopírovat. V tomto bodě jsou literály zpracovány. Vyskytuje se zde testování indexu bloku, zda se již nenachází na konci bloku. Jedná se o implementaci dodatečného omezení od referenčního řešení autora algoritmu, kdy po-

slední sekvence bloku končí literálovou částí. Pokud se podmínka vyhodnotí jako pravdivá, zpracovávání bloku je ukončeno a načte se blok následující.

Pokud však index ještě není na konci bloku, pokračuje se zpracováváním match části. Ukazatel `Match_Offset` odkazuje na aktuální hodnotu offsetu v sekvenci, která se nachází přímo za literály. Ověří se, že offset je menší nebo roven velikosti dat uložených ve vyhledávacím okně. Jinak je offset neplatný a program je ukončen. Nyní se počítá délka shody posloupnosti, ke které je potřebný dříve načtený token, resp. jeho 4 nižší bity. Výsledná délka bude uložena v proměnné `Match_Size`, do které se nejprve uloží zmiňované dolní 4 bity pomocí bitové masky a operace bitového součinu a přičte se k nim implicitní minimální délka shody, tedy 4. Poté se využije stejný `while` cyklus jako u načítání délky literálů. Po zjištění offsetu a délky shody je možné match posloupnost zkopírovat na konec vyhledávacího pole. Vyskytuje se zde však problém s velikostí okna. Shoda může začínat až na samotném začátku okna. Pokud je již okno plné, je nutné ze začátku bajty vysunout a uvolnit tak místo, čímž by ale došlo ke ztrátě dat ke kopírování. Proto je nutné dodatečné pole, díky kterému se této ztrátě předejde. Bajty se nejdříve zkopírují do záložního bufferu, v okně se uvolní patřičná kapacita a následně se data přesunou z bufferu do vyhledávacího okna.

Při kopírování shodné posloupnosti mohou nastat 2 případy. Pokud je její délka kratší nebo rovna hodnotě offsetu, lze tuto posloupnost přímočaře pomocí funkce `memcpy` zkopírovat. V případě, že je ale posloupnost delší než hodnota offsetu, posloupnost přesahuje data ve vyhledávacím okně. V takové situaci kopírování probíhá ve 2 krocích. Nejprve je posloupnost o délce offsetu zkopírována přímo. Poté následuje kopírování bajt po bajtu od začátku nově zkopírované části posloupnosti na její konec. Tuto přesahující část nelze zkopírovat přímo, neboť v případě, že je offset roven 1, není pro rekonstrukci posloupnosti k dispozici více než jeden bajt. Proto zde nemá použití `memcpy` smysl. Tímto končí cyklus zpracovávající komprimovaný blok. Pokud je celý blok zpracován, cyklus skončí a načte se další blok. Dekompresor podporuje bloky ukončené literálovou i match částí. Kompresor, který byl použit pro vytvoření vstupního archivu, tedy nemusí být kompatibilní s referenční implementací.

5.1.2 Optimalizace dekompresoru

Po rychlém testu se ukázalo, že dekomprese Silesia korpusu je přibližně 16x pomalejší v porovnání s referenčním dekompresorem. Po analýze implementace byl zjištěn problém u práce s vyhledávacím oknem. Pokud je okno zaplněné a mají-li se na jeho konec vkládat literály nebo shodné sekvence, je potřeba bajty ze začátku vysunout a zapsat na výstup. Právě tyto manipulace s malými datovými celky jsou zdrojem neefektivity. Velice často se přesouvají a zapisují pouhé desítky bajtů, což z hlediska procesorů a počítačů není doporučený způsob práce s daty.

Jednotlivé paměťové a výstupní operace budou agregovány do větších, čímž by měla klesnout úroveň režie s nimi spojená. Vyhledávací okno bude dvojnásobné – o velikosti 128 KiB. Slovník se bude postupně zaplňovat, až do jeho maximální velikosti. Jakmile je slovník plný a je potřeba do něj vložit další data, celá první polovina slovníku je vysunuta a zapsána na výstupní kanál, neboť data v této části z pohledu match offsetu již nejsou dosažitelná, a proto mohou být hromadně přesunuta. Po vysunutí je druhá polovina okna, která je z hlediska match posloupností stále validní, přesunuta funkcí `memcpy` na začátek okna (do jeho první poloviny). Tímto je četnost manipulací s daty minimalizována a zároveň efektivita jednotlivých operací značně zvýšena. Zvětšením slovníku se také odstraní

zmiňovaný problém se shodnou posloupností v případě, že je slovník plný a hodnota offsetu maximální, neboť první bajt v okně bude stále i po vysunutí dat dostupný. Odpadá tedy potřeba dalšího pomocného bufferu, ale paměťová náročnost zůstává stejná, neboť použitá paměťová kapacita z bufferu se akorát připojila ke slovníku.

5.1.3 Kompresor

U kompresoru není vyžadována žádná míra obecnosti ze strany specifikace algoritmu, takže schopnost dekomprimovat archivy z kompresoru je ponechána čistě na dekompresoru. Není určená žádná minimální konfigurace, kterou by musely kompresor i dekompresor podporovat. Je možné pozorovat asymetrické požadavky na jednotlivé komponenty. Kompresor může být velmi specifický, aby splnil stanovené požadavky pro dané prostředí a aplikaci, čehož se při této implementaci plně využije. Konfigurace LZ4 rámce je staticky určena v programu – maximální velikost dat v bloku 4 MiB, datové bloky jsou závislé na předchozích, velikost původních dat a veškeré nepovinné kontrolní součty jsou vynechány. Povinný kontrolní součet deskriptoru je pro tuto konfiguraci předpočítán a stačí ho pouze vložit za deskriptor.

Datové struktury

V programu se nachází 2 ústřední datové struktury. `Lz4Context`, podobně jako v případě dekompresoru, obsahuje položky určující stav komprese na úrovni datových bloků. Jelikož není dopředu známá velikost vstupních dat, nelze nijak rozhodovat o celkovém stavu. Komponenta `BlockHolder` uchovává aktuální rozpracovaný komprimovaný nebo nekomprimovaný blok. Velikost tohoto pole je 4 MiB plus další 4 bajty, aby kromě dat v bloku bylo možné do pole uložit i velikost bloku, což z pole činí primární zdroj dat k zápisu na výstupní kanál. V komponentě `BlockHolderSize` se nachází velikost aktuálně rozpracovaného bloku, která je po ukončení bloku vložena na začátek pole `BlockHolder`. Ukazatel `BlockHolder_Position` slouží pro pohyb v rozpracovaném bloku a pro postupné ukládání komprimovaných LZ4 sekvencí nebo plnění bloku nekomprimovanými daty. Pole `Buffer` představuje vyhledávací okno. U dekompresoru jsou okno a slovník spojovány, jedná se o tutéž entitu. V případě kompresoru jsou tyto datové struktury odděleny do samostatných entit, které spolu úzce kooperují. Okno je svým způsobem pevně dané, kdežto slovník se může u každého kompresoru lišit – je variabilní. Po zkušenosti s důležitostí velikosti vyhledávacího okna u dekompresoru byla i zde zvolena dvojnásobná velikost, tedy 128 KiB. Čítač `DataSizeToProcess` určuje počet bajtů ve vyhledávacím okně, které ještě nebyly zpracovány. Ukazatel `Buffer_CurrentPosition` odkazuje na pozici ve vyhledávacím okně, kde se nachází další bajt ke zpracování. Ukazatel tedy slouží k postupnému zpracovávání dat v okně. Další čítač `CompressedDataSize` hlídá velikost dat uchovaných v aktuálně rozpracovaném komprimovaném bloku. V případě komprimovaných dat je zcela zřejmé, že velikost bloku bude menší než původní data před samotnou kompresí, a proto tato proměnná určuje velikost dat po expanzi (nebo před kompresí), aby bylo možné zaručit splnění limitu maximálně 4 MiB dat na blok. U nekomprimovaného bloku postačuje `BlockHolderSize`, neboť velikost bloku odpovídá velikosti dat uvnitř bloku. Vnořená struktura `MatchContext` je zapouzdřený slovník, kterému je věnovaná samostatná sekce. Příznak `IsBlockUncompressed` slouží při zápisu bloku na výstup, zda je zapisovaný blok komprimovaný či nekomprimovaný, aby se případně doplnil do velikosti bloku příznak nekomprimovaného bloku. Další příznak `EndOfInput` slouží k přenosu informace ohledně dostupnosti dat ze vstupního kanálu. Nejdou-li další data ze standardního vstupu načítat, nastaví se tento příznak.

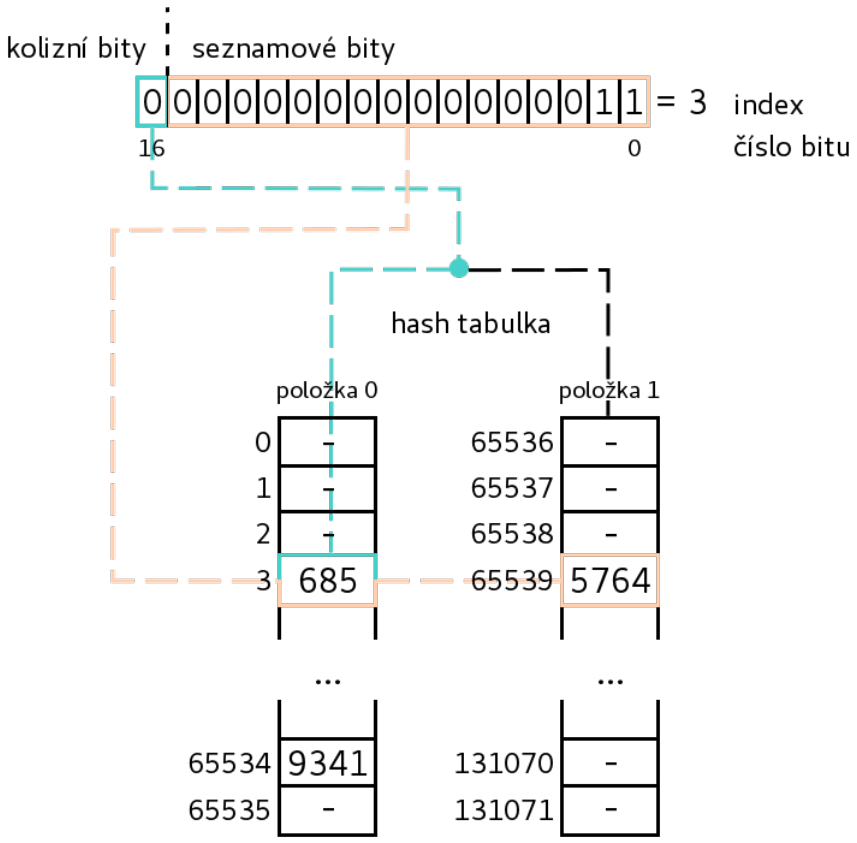
Druhou hlavní strukturou je `BlockPartMetadata`, která obsahuje údaje o kompresi na úrovní LZ4 sekvencí. Jedná se o pouhá metadata, žádná data nejsou v této struktuře uložena. Změna ve strukturách se propaguje ze zdola – při hledání sekvence se mění komponenty `BlockPartMetadata` a při nalezení se změna skokově propaguje do `Lz4Context` ve formě posunutí ukazatelů, přičtení k čítačům a zapsání dat do `BlockHolder`. Struktura obsahuje skupinu proměnných pro práci s literály a match posloupnostmi. `LiteralsLength` určuje počet literálů v aktuální LZ4 sekvenci. `LiteralsLength_Block` slouží pro výpočet a dočasné uložení počtu bajtů s hodnotou 255, které budou zapsány do bloku k určení délky literálů v LZ4 sekvenci při dekompresi. Podobně `LiteralsLength_Bytes` obsahuje hodnotu posledního bajtu (menšího než 255) určující délku literálů. Skupina s metadaty o match posloupnostech obsahuje proměnné `MatchLength`, `MatchLength_Blocks` a `MatchLenght_Bytes` se stejnou funkcí jako u literálů, ale pro posloupnosti. Navíc do této skupiny patří komponenta `MatchOffset` pro uložení offsetu match sekvence od pozice ukazatele aktuálně zpracovávaných dat ve vyhledávacím okně, proměnná `MatchPosition`, která při ověřování, zda jsou posloupnosti shodné, určuje absolutní pozici potenciální match posloupnosti v okně. Dále se ve struktuře nachází ukazatel `Buffer_CurrentPosition`, který slouží na pohyb ve vyhledávacím okně při tvorbě LZ4 sekvence. Po jejím nalezení je hodnota ukazatele použita k přepsání `Buffer_CurrentPosition` o úroveň výš ve struktuře `Lz4Context`. Ukazatel `BlockHolder_TokenPosition` uchovává pozici tokenu pro zapisovanou LZ4 sekvenci, čímž je možné odděleně zapisovat literály a match posloupnost. Položka `BlockPartState` výčtového datového typu má režijní charakter. Informuje program, zda byla část sekvence již zpracována. Jedná se tedy spíše o součást logiky kompresoru.

Slovník

Slovník se nachází zapouzdřený ve struktuře `MatchContext` uvnitř `Lz4Context`. Jak bylo v návrhu popisováno, jedná se o hash tabulku `HashTable` s 65 536 položkami, kde každá položka disponuje kolizním seznamem délky 2. Celková velikost hash tabulky je tedy 131 072 položek. Položka tabulky je typu `HashTableCell_t`. Obsahuje index začátku match posloupnosti ve vyhledávacím okně `Cell` a příznak `IsUsed`, zda je daná položka v tabulce obsazená. V hash tabulce se tedy explicitně nenachází zmiňovaný kolizní seznam. Všechny buňky jsou rovnocenné. Jednotlivé kolizní seznamy a jejich buňky se rozlišují způsobem indexování. V základu je možné hash funkcí získat index v intervalu $< 0, 65\,535 >$, čímž se přistupuje k prvním položkám kolizních seznamů. Velikost tabulky umožňuje uložit i druhé položky, přičemž přístup k jednotlivým položkám je rozlišen způsobem indexování. Pro indexování kolizních seznamů postačuje 16 bitů, avšak pro přístup ke všem položkám tabulky je potřeba 17 bitů. Index lze tedy rozdělit na dvě části. Prvních 16 bitů (seznamové bity) identifikuje kolizní seznam a nejvyšší 17. bit (kolizní bit) určuje danou buňku seznamu, neboť tabulku rozděluje na dvě oblasti o stejné velikosti. Princip fungování této úpravy hash tabulky ilustruje obrázek 5.1. Výhodou přistupování k položkám tímto způsobem je snadná rozšiřitelnost. Je-li počet bitů identifikující položky seznamu roven n, získáme kolizní seznamy o délce až 2^n položek. Kolizní bity nemusí být nutně využity všechny, ale proměnná, pomocí které se hash tabulka indexuje, musí bitovou délku dodržet, jinak některé buňky seznamů nebudou přístupné.

Inicializace

Nejdříve se inicializuje struktura `Lz4Context`. Většina komponent může být nastavena na 0, kromě `Buffer_CurrentPosition`, které je nastaven odkaz na začátek vyhledávacího okna,



Obrázek 5.1: Princip indexace položek hash tabulky v kompresoru

a `BlockHolder_Position`, která odkazuje 5. bajt v poli `BlockHolder`. 4 bajty jsou rezervovány pro zápis velikosti bloku, než je blok přeposlán na výstup. Poté se program pokusí naplnit celé vyhledávací okno vstupními daty, tedy načíst funkcí `fread` 128 KiB ze standardního vstupu a velikost načtených dat uložit do `DataSizeToProcess`. Pokud již v tento moment `fread` vrátí méně dat než bylo požadováno, veškerá data ze vstupu byla načtena, a proto se nastaví příznak `EndOfInput`. Je-li vstup nenulový, zapíše se na výstup předpřipravená statická hlavička s magickou konstantou a deskriptorem LZ4 rámce. Pokud se žádná data na vstupu nenachází, program je ukončen, aniž by byla vyprodukována jakákoliv data. Druhá možnost by byla vygenerovat prázdný archiv, který je také validní. Poté se inicializují složky struktury `BlockPartMetadata` na 0, až na `Buffer_CurrentPosition`, která se odkazuje na začátek vyhledávacího okna jako stejnojmenná proměnná v `Lz4Context`.

Logika a struktura programu

Jádro kompresoru představuje funkce `compressData` s parametry `Lz4Context` a `BlockPartMetadata`, ve které probíhá samotné vyhledávání match sekvencí a práce se slovníkem. Svým způsobem řídí celý chod programu. Podle různých situací vrací funkce příslušné hodnoty z výčtového typu `CompressReturnValues_t`, na základě kterých se ve `switch` konstrukci vykonávají nutné operace. Volání funkce a `switch` jsou zapouzdřeny ve `while` smyčce, která se opakuje, dokud nejsou zpracována všechna data ze vstupu, resp. dokud není složka `DataSizeToProcess` rovna 0. Jakmile jsou data zpracována, rozpraco-

vaný blok je ukončen a zapsán na výstup. Za ním následuje ukončovací značka, tedy nulová velikost bloku, a program je ukončen.

Funkce `compressData` se skládá ze dvou částí. Jejich využití se navzájem vylučují. První část se provádí pouze v případě opakovaného volání funkce s rozpracovanou LZ4 sekvencí. Tato část umožňuje pokračovat v porovnávání nalezené match posloupnosti a aktuálních vstupních dat v případě, že ve vyhledávacím okně byla načtená data vyčerpána. Další možnost by byla posloupnost, a tím i LZ4 sekvenci, ukončit, čímž by se však zhoršoval dosažitelný kompresní poměr, a tím i potenciál metody. Druhá část sestavuje LZ4 sekvenci od začátku. Nejprve se zkontroluje, že velikost nezpracovaných dat ve vyhledávacím okně je vetší než 12 bajtů. Tato podmínka vychází z dodatečných omezení referenční implementace zmiňovaných ve specifikaci algoritmu. Pokud je v okně dat méně, funkce je s návratovou hodnotou ukončena, odpovídající `case` ve `switch` příkazu zareaguje a program se pokusí do vyhledávacího okna načíst další vstupní data. Jednotlivé návratové hodnoty funkce lze považovat za druh příkazů. `compressData` vrátí pokyn k načtení dat pouze pokud není nastaven příznak `EndOfInput`, neboť jinak již nejsou další data dostupná. V takovém případě se zbytek dat připojí k literálům a přejde se k zapisování poslední sekvence a bloku archivu.

Pokud je podmínka o velikosti načtených dat splněna, vstoupí se do dalšího `while` cyklu, ve kterém probíhá proces sestavování LZ4 sekvence. Opakování cyklu nastává při přidávání literálů do vytvářené sekvence. Zopakování cyklu závisí na podmínce, která kontroluje, zda je velikost nezpracovaných dat ve vyhledávacím okně větší než 5 bajtů. Tato podmínka se opět odvíjí od dodatečných omezení referenční implementace, která uvádí, že minimálně posledních 5 bajtů musí být v bloku zapsáno jako literály. Nesplněním podmínky cyklus končí a program se pokusí načíst další data do okna stejným způsobem, jako v případě předchozí podmínky ohledně 12 bajtů. Může nastat situace, kdy data byla do okna načtena v pořádku bez detekce konce vstupních dat, ale žádná další data se na vstupu nenachází. Taková situace nastane v případě, kdy velikost načítaných dat přesně odpovídá objemu dostupných dat na vstupním kanále. Nelze tedy v tomto případě spoléhat na příznak `EndOfInput`, neboť tuto situaci nedetekuje. Proto je nutné zbývající počet bajtů v okně kontrolovat neustále, aby v případě, že se další data nepodaří načíst, bylo vždy alespoň 5 bajtů dostupných pro řádné ukončení bloku podle referenční implementace.

Jsou-li datové prerekvizity splněny, přechází se na samotné vyhledávání sekvencí. Nejprve se vypočítá hash 3 vstupních bajtů pomocí funkce `hashTable_hash` (v tomto případě CRC), čímž se získá index první položky kolizního seznamu uloženého v hash tabulce. Nyní následuje řada větvení. Není-li první buňka v tabulce využita (příznak `IsUsed` není nastaven), znamená to, že celý kolizní seznam je prázdný a není ho tak nutné dále procházet, neboť seznamy se zaplňují vždy směrem od první položky. V takovém případě je příznak nastaven a do buňky je vložena hodnota indexu, kde hledaná trojice bajtů v okně začíná. Jelikož je vyhledávací okno velké 128 KiB, indexy do okna jsou až 17bitové. Poté je první bajt z hledané posloupnosti zařazen k literálům aktuálně sestavované LZ4 sekvence. Zvýší se čítač `LiteralsLength`, o jedna se sníží velikost dat v `DataSizeToProcess` a posune se ukazatel `Buffer_CurrentPosition` ve struktuře `BlockPartMetadata`, neboť se pracuje na nižší úrovni LZ4 sekvencí. Cyklus `while` se pak opakuje. Pokud počet literálů přesáhne určitou mez, konkrétně 65 535 literálů, bude rozpracovaný blok ukončen a literály se vloží do nového nekomprimovaného bloku. Velikost této hranice je stanovena opět na základě způsobu práce s vyhledávacím oknem.

V případě, že příznak `IsUsed` je v indexované položce hash tabulky nastaven, nachází se v buňce dříve zapsaný index do vyhledávacího okna, u kterého je nutné v první řadě prověřit jeho platnost. Do proměnné `MatchPosition` je přiřazen odkaz odpovídající uloženému in-

dexu v hash tabulce přičtením tohoto indexu k adrese začátku vyhledávacího okna. Jelikož je velikost okna dvojnásobná, je nutné zkontrolovat, zda se potenciální shodná posloupnost bajtů nachází v dosahu maximální povolené hodnoty offsetu. Jinak řečeno, zda vzdálenost odkazovaného začátku posloupnosti od zpracovávaného bajtu je menší nebo rovna 65 535. Zároveň se nalezený index musí nacházet před zpracovávaným bajtem. Tato podmínka vyplývá ze způsobu práce s vyhledávacím oknem, který je rozebrán v samostatné části. Nejsou-li tyto podmínky splněny, jedná se o nevalidní index. V takovém případě je index odstraněn, konkrétně je buňka přepsána druhou položkou v kolizním seznamu, která může obsahovat nějaký index nebo je prázdná, a původní druhé položce je nastavena neaktivita. Poté se proces kontroly položky opakuje, čímž se v podstatě kontroluje druhá položka na pozici první. Pokud je nevalidní buňka druhou v kolizním seznamu, je index pouze přepsán a bajt přidán k literálům. Tato situace však nastává pouze ve speciálním případě.

Pokud index podmínkám vyhovuje a je tedy validní, přechází se na kontrolu samotné posloupnosti. Do položky `MatchOffset` se přiřadí offset indexu od aktuální pozice ve vyhledávacím okně v `BlockPartMetadata` jako rozdíl ukazatelů. Dále se již nachází porovnávání jednotlivých bajtů. Je ale nutné rozlišovat dva případy. Pokud je součástí rozpracované LZ4 sekvence méně jak 5 literálů, volá se procedura `matchCheck`, která porovnává maximálně 4 bajty posloupnosti se zpracovávanými bajty. Pokud si bajty odpovídají, postupně se navyšuje čítač `MatchLength`. `MatchPosition` se využívá pro posun ze strany posloupnosti a lokální proměnná pro posun ve vstupní části. Pokud některá z dvojic bajtů nesouhlasí, položka `MatchLength` je vynulována a procedura se vrací. Jinak v položce struktury zůstává uchována délka posloupnosti 4. Je-li literálů v sekvenci více než 5, volá se funkce `findMatch`, která má velmi podobnou funkci jako `matchCheck`. Není však limitovaná délkou posloupnosti 4 a vrací hodnoty z výčtového datového typu `FindMatchReturnValues_t`, neboť je třeba hlídat některé situace. Uvnitř funkce může při porovnávání bajtů klesnout počet dostupných dat na 5 bajtů. V takovém případě je potřeba zkusit načíst další data, neboť zbylé bajty jsou potřeba k dříve popsanému korektnímu ukončení bloku. V takovém případě se vrátí patřičná hodnota a příkaz k načtení dalších dat ze vstupu je propagován programem dál, není-li nastaven příznak `EndOfInput`. Po načtení dat se znovu volá funkce `compressData`, ale s rozpracovanou LZ4 sekvencí, tudíž se provede první část funkce, jak bylo popsáno výše. Pokud posloupnost dosáhne alespoň délky 4, není čítač `MatchLength` vynulován.

Důvodem, proč je hledání match sekvencí takto rozděleno, je dodatečné omezení referenční implementace vyžadující přítomnost alespoň 5 literálů na konci komprimovaného bloku. V situaci, kdy by sekvence neobsahovala žádné literály, pouze match posloupnost, a celá sekvence by se do aktuálně rozpracovaného bloku nevešla, je nutné tento blok ukončit a sekvenci vložit do nového bloku. Jenže pokud nejsou k dispozici žádné literály, nelze blok správně ukončit. Match posloupnost by se musela rozbíjet, čímž by mohly vzniknout další komplikace, např. u match sekvence délky 4 – posloupnost by se musela celá zrušit. Z režijního pohledu se s již sestavenou LZ4 sekvencí nevyplácí příliš manipulovat. Proto je uměle zavedeno omezení, že každá sekvence při vytváření musí obsahovat alespoň 5 literálů.

Po návratu z jednoho z podprogramů se zkontroluje čítač `MatchLength`. Je-li nulový, nebyla nalezena match sekvence alespoň minimální délky 4. Přechází se na druhou položku v kolizním seznamu nastavením 17. bitu hash hodnoty na 1 a celý proces se opakuje. Pokud se jedná již o druhou položku, je přesunuta na pozici první položky a do druhé je vložen aktuální index okna. Jinak řečeno, starší buňka je přepsána druhou, na jejíž místo je vložen nový aktuální index ve vyhledávacím okně. V případě, že je čítač `MatchLength` nenulový, zkontroluje se počet literálů. Je-li jich méně jak 5, tak se vstupní bajt zařadí k literálům

a čítač se vynuluje. Jedná se tedy pouze o kontrolu validity hodnot a posloupností ve slovníku. Pokud sekvence obsahuje větší počet literálů, je sestavení sekvence potvrzeno návratovou hodnotou, a sekvence tak může být zapsána do rozpracovaného bloku. Zapisovat se může sekvence celá, nebo pouze její match část, neboť literály byly zapsány již dříve. Důvod je vysvětlen u popisu práce s vyhledávacím oknem.

Vyhledávací okno a `switch` konstrukce

Vyhledávací okno funguje podobně jako u dekompresoru, u kterého byla kvůli efektivitě práce se vstupním kanálem a manipulací s pamětí nastavena velikost vyhledávacího okna na 128 KiB. Stejná velikost okna je proto použita i v rámci kompresoru. Nejprve se celé okno zaplní daty. Jakmile se ukazatel aktuální pozice přesune na konec okna, je druhá polovina okna přesunuta funkcí `memcpy` do první poloviny a uvolněná druhá polovina je naplněna daty novými. Přesouvání dat zajistí, že bajty v dosahu offsetu od ukazatele `Buffer_CurrentPosition` zůstanou dostupné. S touto změnou pozice také souvisí indexy uložené ve slovníku. Pro efektivní využití slovníku je nutné tyto přesuny reflektovat, takže po každém přesunu je procedurou `updateHashTable` invertován 17. bit u všech položek ve slovníku, čímž se zajistí platnost dosažitelných indexů a zneplatnění těch, které odkazovaly na posloupnosti nacházející se dříve v první polovině vyhledávacího okna. Proto se u kontroly validity vyskytuje podmínka, že index ze slovníku musí být nižší než index aktuálně zpracovávaných dat. Vyšší indexy jsou zneplatněné dřívější odkazy.

Při přesouvání je ale nutné si dávat pozor, neboť může nastat situace, kdy se přepíšou data, která ještě nejsou zapsána v bloku. Aby se těmto ztrátám předešlo, je nastaven maximální počet literálů na 65 535. Při překročení hranice se literály stanou součástí nově vytvořeného nekomprimovaného bloku. V situaci, kdy máme 65 540 literálů a zpracovávající ukazatel je již na konci vyhledávacího okna, je nutné provést přesun. Jelikož se ale druhá polovina dat přesouvá na pozici první, došlo by k přepsání a ztrátě 4 bajtů, jelikož struktura `BlockPartMetadata` literály fyzicky neobsahuje, pouze odkazuje na jejich pozice ve vyhledávacím okně. Podobný případ může nastat při porovnávání nalezené match posloupnosti a vstupních bajtů. Pokud je posloupnost delší jak 4 bajty a je potřeba načíst další data ze vstupu, literály se před načtením dat zapíší do rozpracovaného bloku, neboť posloupnost může být teoreticky velmi dlouhá a po přesunech by tak mohlo dojít opět ke ztrátě literálů. Proto je ve `switch` části oddělený zápis literálů a match posloupnosti. Hodnota `BlockPartState` v `BlockPartMetadata` pak určuje, zda byly literály již zapsány či nikoliv.

Části `switch` konstrukce zajišťují zmiňovaný zápis literálů a match posloupností do bloku, přesun stávajících a načtení nových dat do vyhledávacího okna, ukončení rozpracovaného bloku s 5 literály a zápis zbytku literálů do nového nekomprimovaného bloku.

5.1.4 Optimalizace kompresoru

Krátký test implementace kompresoru na Silesia korpusu ukázal, že komprese je mnohonásobně pomalejší než v případě referenčního řešení. Profilováním programu bylo zjištěno, že úzkým hrdlem je funkce počítající CRC `hashTable_hash`. Výpočet je postaven na dvou `for` cyklech. Vnější cyklus iteruje nad bajty a vnitřní nad bity aktuálního bajtu. Celkem tedy proběhne 24 iterací při každém volání funkce. Kromě výpočtu po bitech existují i další způsoby po větších celcích. Tabulková metoda [13] umožňuje výpočet CRC po bajtech, čímž by se snížil počet iterací, a tím urychlila funkce. Tabulka o velikosti 256 položek obsahuje předpočítané hodnoty závislé na konkrétním typu CRC a inicializačním polynomu, které

jsou adresovány v cyklu během výpočtu. Tabulka hodnot spolu s příslušným algoritmem byli vygenerovány pomocí programu *pycrc* [9].

Dále byla upravena hranice počtu literálů pro tvorbu nekomprimovaného bloku. Ukončení rozpracovaného bloku a založení nového znamená vložení 5 literálů na konec a zapsání bloku. Z hlediska metadat se tedy jedná o dodatečný zápis tokenu a velikosti bloku. Celkem tedy 5 bajtů metadat navíc. Hledá se takový počet literálů, pro který je výhodnější vytvořit vlastní blok, než je zapisovat s metadaty do stávajícího komprimovaného bloku. Mohou-li metadata popisující počet literálů obsazovat maximálně 5 bajtů, nová hranice vychází na $15 + 4 \cdot 255 + 254 = 1\,289$ literálů. Po aplikování změny hranice klesla velikost výsledného archivu se Silesia korpusem o 4 931 bajtů. Efektivita této uprávy vyniká především u težko komprimovatelných dat, u kterých je vysoký počet literálů.

5.2 Hardwarová verze

K implementaci dekompresoru a kompresoru v hybridní či syntetizovatelné formě je použit nástroj Catapult, ve kterém je dostupný programovací jazyk C++ v omezené podobě. Softwarová verze zde slouží jako předloha. Obě jednotky budou také pracovat na úrovni archivů. Kompresor transformuje ucelený kus dat na LZ4 archiv. V případě dekompresoru není příliš velký rozdíl mezi datovým rámcem a celým archivem, jelikož za sebou jdoucí rámce v jednom archivu jsou nezávislé, a proto je lze považovat za samostatné archivy. Mezi markantní rozdíly od softwarové verze patří způsob práce s pamětí. Veškeré proměnné a pole musí být alokované staticky. Dále nelze využít funkci `memcpy`, protože není možné přetypovat ukazatel. Z toho vyplývá, že nelze manipulovat s celými částmi polí. Je tedy nutné s poli pracovat na pouze úrovni jednotlivých položek. Použití ukazatelů v programu tedy ztrácí svůj hlavní smysl, a veškeré ukazatele lze tudíž nahradit indexy, což je pro prostředí FPGA čipů přirozenější.

Další velkou změnou je rozhraní obou modulů. Jak bylo popisováno v návrhu, standardní vstup a výstup není součástí FPGA čipů. Vstupní a výstupní kanál bude vytvořen pomocí rozhraní AC Channel popsaného v sekci 4.3.2. Paralelní načítání více bajtů nemá využití, pokud jednotky nejsou schopny bajty paralelně zpracovat. Sériové rozhraní se také velmi hodí ke zmiňovanému způsobu práce s poli a pamětí obecně. S rozhraním také souvisí chybové návratové kódy. V FPGA běží program neustále, nikdy není ukončen. Z toho vyplývá, že není možné použít návratové kódy pro předání informace o výskytu chyby, nýbrž je potřeba speciální management kanál, kudy budou tyto informace proudit. Jelikož se výstupní data z jednotek nemusí ukládat lokálně a mohou být zasílána přímo do sítě, je nutné v případě výskytu chyby uprostřed zpracovávání informovat přijímající stranu o této skutečnosti, aby již přijatá data mohla být invalidována. Tento kanál však není součástí implementace.

Knihovna Algorithmic C [8], dostupná v nástroji Catapult, nabízí celočíselný datový typ s bitovou přesností pomocí C++ šablony `ac_int<bit_width, is_signed>`, kde `bit_width` je počet bitů typu a `is_signed` je příznak, zda je datový typ znaménkový či nikoliv. Díky této šabloně lze vytvořit např. 17bitovou proměnnou, aniž by se plýtvalo přebytečnými bity, které by jako v případě softwarové verze nebyly využity. Pomocí statické funkce `ac::nbits<`x`>::val` lze při překladu programu zjistit počet bitů, který je nutný pro reprezentaci hodnoty x. Podobně statická funkce `ac::log2_ceil<`x`>::val` při překladu vypočítá funkci $\log_2(x)$ zaokrouhlenou nahoru, což je vhodné pro výpočet minimálního počtu bitů nutných pro adresaci x položek. Obě funkce jsou používány při deklaraci některých proměnných s bitovou přesností, aniž by bylo potřeba počet bitů ručně počítat.

Jelikož byla implementace kompresoru i dekompresoru podrobně popsaná v softwarové pasáži, budou v této sekci uvedeny pouze odlišnosti struktury programů a důležitých operací.

5.2.1 Dekompresor

Díky změně práce s vstupním a výstupním kanálem se určitým způsobem zjednodušila práce se slovníkem. Jelikož se u polí pracuje pouze s jednotlivými položkami, není důvod k vytvoření většího vyhledávacího okna než je nezbytně nutné, neboť zde neplatí stejné principy efektivní práce s pamětí jako u klasických počítačů. Okno je však přece jen o 1 bajt zvětšené. Indexy vyhledávacího okna jsou tak v rozsahu 0 až 65 535, je tedy využit plný rozsah hodnot 16bitové proměnné, díky čemuž je možné se slovníkem pracovat jako s kruhovým bufferem bez nutnosti operace modulo. Nová dekomprimovaná data automaticky přepisují stará a není nutné provádět žádné přesuny jako v případě softwarové verze, čímž kompletně odpadají režijní nároky slovníku.

S rozhraním také souvisí nová komponenta `ByteHolder` ve struktuře `Lz4Context`. Veškerá dekomprimovaná data se rozdvojují – jsou zapsána na konec vyhledávacího okna a na výstupní kanál. Nelze dopředu určit, zda poslední dekomprimovaný bajt z bloku je zároveň poslední bajt dat v celém archivu. Rozhraní však vyžaduje znalost, zda je zapisovaný bajt poslední. Proto je nutné bajt před výpisem na výstup pozdržet, dokud není známo, zda se opravdu o poslední bajt jedná či nikoliv. `ByteHolder` je tedy v podstatě zpožďovací buffer o velikosti 1 bajtu, který je na konci dekomprese archivu vyprázdněn.

Protože se data načítají ze vstupu po bajtech, nemá smysl nejdříve celý datový blok načíst a až poté začít se zpracováváním, neboť načítání velkých kusů dat po malých jednotkách není příliš efektivní a rapidně by vzrostla latence celého řešení. Proto se bloky zpracovávají postupně, takže pole pro uchování bloku `BlockHolder` není potřeba, čímž se minimalizují paměťové požadavky dekompresoru. Čtení větších celků dat ze vstupu, např. literálů, je nutné provádět v cyklech.

Struktura programu je mírně pozměněna konstrukcí `switch`. Na základě načtené magické konstanty se rozlišují tři případy – LZ4 rámec, uživatelský rámec a neznámá data. V `case` oblasti pro uživatelský rámec je načtena jeho velikost a postupným načítáním v cyklu je rámec přeskočen. V případě neznámých dat je postup velice podobný. Data jsou opakovaně načítána, dokud načtený bajt nemá nastavený příznak `IsLastByte` a je tedy poslední v aktuálně zpracovávaném datovém celku. Část, ve které se zpracovává LZ4 rámec se nijak zásadně neliší. Nejdříve se zpracuje deskriptor rámce. Poté se opakovaně načítá velikost bloků. Pokud je blok nekomprimovaný, tak jsou v jednom cyklu data uvnitř zkopírována do okna (pokud je v deskriptoru nastavena závislost bloků) a na výstup přes buffer `ByteHolder`. V případě komprimovaného bloku jsou cyklicky zpracovávány LZ4 sekvence, dokud není takto celý blok zpracován. Blok může končit standardně match sekvencí nebo literály jako u referenční implementace.

Nástroj Catapult automaticky zajistí, že pole jsou na začátku běhu programu v FPGA ve smyčce nulována. U vyhledávacího okna `Window` však nulování není vyžadováno, takže je pomocí funkce `ac::init_array<AC_VAL_DC>` nástroji sděleno, že inicializační smyčku nemá vytvářet, čímž se urychlí start dekompresoru.

5.2.2 Kompresor

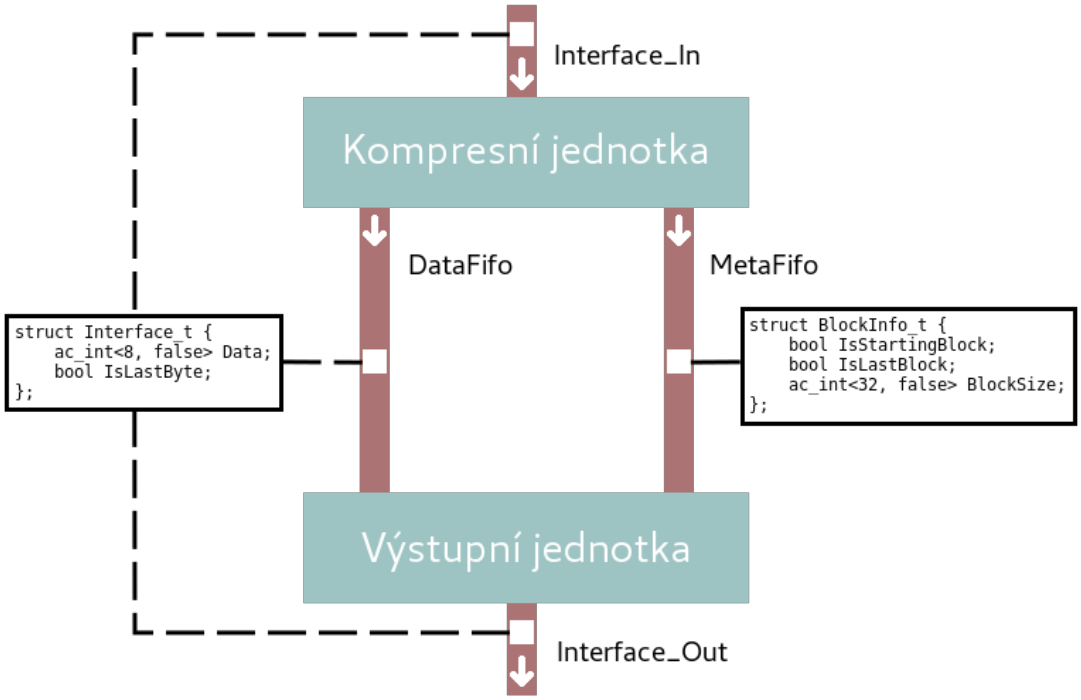
Aby mohl být komprimovaný či nekomprimovaný blok zapsán na výstup, musí být známá jeho velikost. Kompresor tedy uchovává celý blok, dokud není ukončen. Kvůli sériovému

rozhraní by musel být blok na výstup zapisován v cyklu po jednotlivých bajtech, přičemž by žádná jiná akce kromě zápisu neprobíhala. Proto je vhodné zápisovou a výpočetní část oddělit do samostatných jednotek pomocí hierarchického návrhu, což umožní kompresoru zpracovávat vstupní data i během zápisu na výstup. Kompresor se tedy skládá z kompresní a výstupní jednotky, které jsou zobrazeny na blokovém schématu 5.2. Kompresní jednotka ze vstupního rozhraní `Interface_In` načítá postupně data. Veškerá výstupní data (hlavička archivu, data uvnitř bloku) přeposílá výstupní jednotce přes rozhraní `DataFifo` stejného typu jako vstupní či výstupní kanál. Výstupní jednotka si tato data ukládá do interního kruhového bufferu `BlockHolder` o maximální velikosti bloku, tedy 4 MiB plus dalších 11 bajtů navíc pro hlavičku archivu a poslední nulovou velikost bloku. Jakmile má být blok ukončen, kompresor druhým komunikačním kanálem `MetaFifo` pošle výstupní jednotce velikost tohoto bloku a zda se jedná o první nebo poslední blok v archivu. Jakmile výstupní jednotka tyto informace přijme, vloží na výstupní rozhraní `Interface_Out` nejprve velikost bloku a poté začne postupně vysouvat daný počet bajtů z interního pole `BlockHolder`. Pokud je blok startovní, je navíc vysunuto 7 bajtů hlavičky archivu. Je-li naopak blok poslední, po vypsání posledního bajtu z bufferu jsou na výstupní kanál vygenerovány 4 bajty nulové velikosti bloku, čímž je archiv ukončen. V mezičase může kompresní jednotka již zpracovávat další vstupní data a přeposílat je výstupní jednotce, která je implementovaná takovým způsobem, aby zvládala v jednom hodinovém taktu vstupní bajt načíst a vložit do pole a zároveň bajt z pole vložit na výstupní kanál. Zápis do interního bufferu jednotky je možný pouze pokud disponuje volnou kapacitou. Mohla by nastat situace, že poslední blok archivu byl vypsán a pole je již plně obsazeno hlavičkou a blokem z následujícího archivu. Jelikož se nulová velikost bloku generuje, žádné položky pole by se neuvolňovaly, takže by načítání dalších bajtů z rozhraní muselo být pozdrženo, čemuž lze zmiňovaným zvětšením velikosti pole o 4 bajty předejít. Veškerá rozhraní jsou realizována pomocí FIFO front AC Channel, takže díky handshake protokolu je zajištěna synchronizace čtení a zápisu dat do rozhraní. Navíc tak mohou jednotky pracovat na rozdílných frekvencích. Jelikož je výstupní jednotka značně jednodušší, může operovat na vyšších frekvencích, a tudíž by neměla kompresní jednotku nijak brzdit.

Struktura kompresní jednotky prošla významnou transformací. U softwarové verze funkce `compressData` v podstatě řídila celý proces. V této hybridní verzi přebírá řízení konstrukce `switch`, která vytváří stavový automat. Důvodem této změny jsou rozsáhlá větvení, ve kterých se odehrává drtivá většina kódu programu. V nástroji Catapult není doporučováno ve větvích podmíněných příkazů provádět složitější operace, nýbrž je vytknout před nebo za samotné větvení, protože obě větve bývají prováděny paralelně a výsledek je vybrán na základě vyhodnocení podmínky větvení. Jednotlivé stavy automatu částečně složitou hierarchii podmíněných příkazů rozbijí do oddělených celků, což by mělo vést k jednodušší výsledné syntéze obvodu a snížení jeho náročnosti na zdroje čipu. Každý oddělený stav má jasně definovanou funkci, což do jisté míry zjednodušuje strukturu kódu a přispívá k jeho přehlednosti.

Vyhledávací okno je podobně jako v případě dekompresoru zmenšeno na 65 535 bajtů plus dodatečných 12 bajtů kvůli potřebě tyto bajty dopředu načíst ze vstupu. Jedná se o dodatečné omezení referenční implementace, kdy data o velikosti menší než 13 bajtů nelze komprimovat. I v tomto případě funguje okno jako kruhový buffer, ale už s pomocí operace modulo, neboť k dosažení stejného efektu jako u dekompresoru by bylo nutné ponechat vyhledávací okno v původní velikosti.

Jelikož je vyhledávací okno o polovinu menší, je nutné mnohem více hlídat hrozící přepisy odkazovaných literálů během sestavování LZ4 sekvence. Proto byla přidána nová



Obrázek 5.2: Komponenty hybridní implementace kompresoru

komponenta `Literals` do struktury `BlockPartMetadata`. Jedná se o pole pro ukládání nalezených literálů, čímž se zamezí potenciálním ztrátám dat přepisováním kruhového vyhledávacího okna. Pole je schopno uchovat všech 1 289 literálů, což je maximální hranice před vytvořením nekomprimovaného bloku. Je-li v LZ4 sekvenci tato hranice překročena, aktuální blok je ukončen zapsáním 5 literálů z pole, výstupní jednotce je zaslána velikost bloku a zbylý obsah pole `Literals` je přesouván do interního pole ve výstupní jednotce. Veškeré následující literály nejsou udržovány v poli, ale rovnou odesílány, až do okamžiku, než je nalezena match posloupnost. Poté je nekomprimovaný blok ukončen a samotná match posloupnost je odeslána do nově založeného komprimovaného bloku. Aby nebylo třeba rozlišovat případ, jestli byl při tvorbě nekomprimovaného bloku předchozí blok ukončen (pokud je aktuální blok ještě prázdný, lze literály přesunou přímo), je do struktury přidána další položka `LiteralsStartPosition`, která určuje počáteční pozici literálů v poli `Literals`. Pokud byl předchozí blok ukončen, stačí tento index zvýšit o 5. Po zapsání všech literálů je index automaticky resetován. Během tvorby nekomprimovaného bloku a přesouvání literálů se žádné vstupní bajty nezpracovávají ani nenačítají. Ze struktury `BlockPartMetadata` byla komponenta `Buffer_CurrentPosition` odstraněna, aktualizuje se stejnojmenná proměnná přímo v `Lz4Context`.

Automatické nulování pole je funkcí `ac::init_array<AC_VAL_DC>` vypnuto u vyhledávacího pole `Buffer` a u dočasného bufferu pro literály `Literals`, neboť před čtením obsahu obou polí je do nich zapisováno. U interního pole `BlockHolder` ve výstupní jednotce nelze inicializaci zabránit, neboť se jedná o pole struktur, což funkce `init_array` nepodporuje. Naopak v případě slovníku je žádoucí příznaky `IsUsed` u buněk před začátkem každé komprese inicializovat na nuly.

U softwarové verze bylo načítání dat ze vstupu centralizováno do jedné oblasti, neboť se načítaly velké kusy dat, s čímž byla spojena režie vyhledávacího okna. V této verzi je řízení decentralizováno do jednotlivých stavů automatu. Protože odpadá režie slovníku, je možné načítání dat rozvést do všech potřebných částí programu. Není vytvořený žádný stav pouze pro načítání ze vstupu, ale každá část používá rozhraní dle potřeby.

5.2.3 Syntéza

Transformace dekompresoru na VHDL kód proběhla v pořádku. Smyčky, u kterých je dopředu známý počet iterací, byly plně rozbaleny. Ostatní smyčky, u kterých není počet iterací známý a zároveň neobsahují žádné další vnořené smyčky, byly zřetězeny s inicializačním intervalem (II) 1. Kromě smyčky načítající offset match posloupnosti, u které bylo kvůli datovým závislostem nutné nastavit interval II na 2. Vyhledávací okno `Window` bylo namapováno na dvouportovou blokovou RAM paměť přítomnou na čipu. Ostatní proměnné se nachází v registrech. Obvod se podařilo naplánovat a úspěšně vysyntetizovat při jeho maximální pracovní frekvencí 125 MHz. Využití zdrojů čipu dekompresorem je zobrazeno v tabulce 5.1.

U kompresoru se postupovalo se smyčkami stejným způsobem. Inicializační interval u zřetězených smyček je také nastaven na 1, až na `while` cyklus ve funkci `findMatch`, který porovnává jednotlivé bajty posloupností, kde bylo kvůli datovým závislostem nutné nastavit II na 4. U mapování však nastal problém se zdroji. Nástroj Catapult neumožňuje mapovat proměnné na dostupné LUT, pouze na registry, blokovou RAM a případně ROM paměť. Tabulka hodnot pro výpočet CRC se chová jako paměť ROM, ale kvůli častému přístupu to tabulky je vhodné umožnit rychlý přístup k položkám, takže jsou tabulce přiděleny registry. Vyhledávací okno `Buffer`, pole `Literals`, celý slovník `MatchContext` a interní pole výstupní jednotky `BlockHolder` jsou namapovány na dvouportovou paměť RAM. Celkově je tedy potřebná RAM přibližně o velikosti 4 961 KiB, což přesahuje kapacitu 4 230 KiB na cílové FPGA platformě Virtex-7 H580T, a proto výsledná syntéza skončí chybou. Z důvodu nepoužitelnosti LUT je tedy nutné snížit paměťovou náročnost kompresoru, čehož je dosaženo snížením maximální velikosti dat v bloku podle tabulky 3.1 na 1 MiB. Z redukce velikosti bloku také vyplývá přepočítání staticky nastaveného deskriptoru, které není striktně vyžadováno, ale je doporučeno. Jelikož bude blok maximálně 1 MiB velký, lze patřičně zmenšit velikost interního pole výstupní jednotky, čímž se požadavky kompresoru na paměť RAM zredukují na přibližných 1 505 KiB a cílová platforma je schopna jim vyhovět. Po úpravě již syntéza proběhla v pořádku a kompresní jednotka je schopna operovat na maximální frekvenci 90 MHz. Spotřeba zdrojů FPGA čipu hardwarovou implementací kompresoru je uvedena v tabulce 5.1.

	Kompresor	**Dekompresor**
LUT	6383 (1,46 %)	1450 (0,33 %)
Registry	3634 (0,42 %)	1053 (0,12 %)
Bloková RAM paměť	342 (36,38 %)	16 (1,70 %)

Tabulka 5.1: Využití zdrojů FPGA čipu Virtex-7 H580T, v závorce je uvedené procentuální využití daného zdroje

Kapitola 6

Srovnání jednotlivých implementací algoritmu LZ4

Porovnávání softwarových implementací bude opět probíhat na RAM disku na superpočítači Salomon. V případě hybridní implementace je pro kompresor i dekompresor připravený test nazývaný testbench. Jedná se o program v jazyce C++, který obaluje implementovaný modul v nástroji Catapult a umožňuje předkládat modulu vstupní data přes jeho definované rozhraní. Na základě těchto testů jsou automaticky vygenerovány podklady pro simulaci výsledného VHDL kódu pro nástroj ModelSim[1]. Testbench je také možné přeložit pomocí klasických překladačů pro procesorové architektury, díky čemuž lze provést funkční verifikaci již na softwarové úrovni.

Další benefit testbenchů představuje možnost některé parametry hybridní verze jednoduše porovnat se softwarovou předlohou. Obecně obě verze nelze na softwarové úrovni přímočaře srovnávat, neboť si nejsou rovnocenné. Druhá verze obsahuje upravené operace a konstrukce, které jsou určené a vyladěné hlavně pro běh na FPGA. Softwarová část se naopak skládá z variant konstrukcí optimálních z hlediska procesorů a práce s operační pamětí.

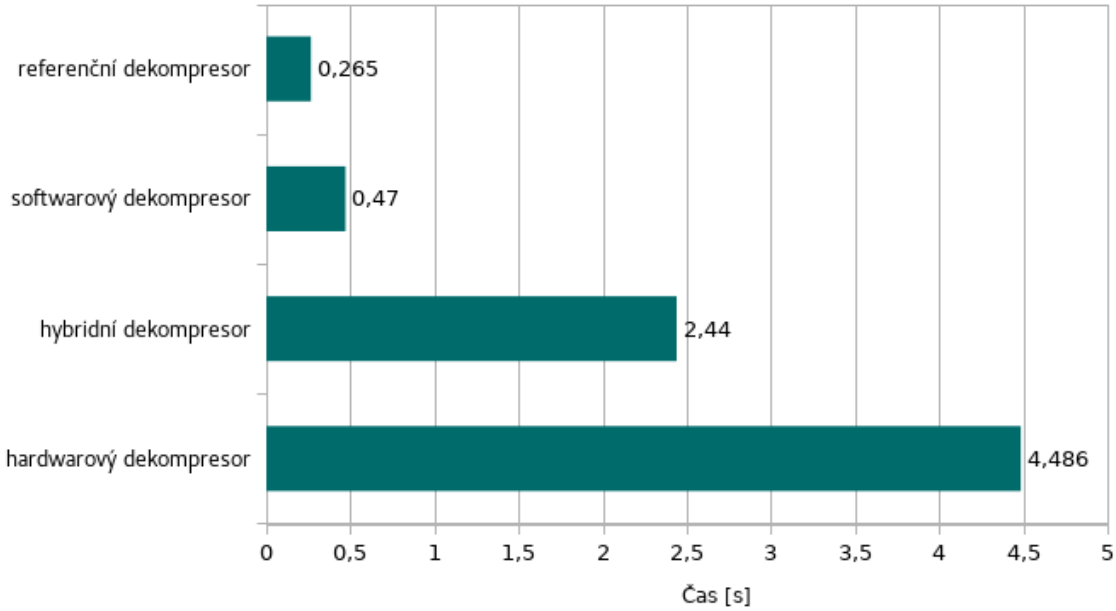
Hybridní implementace tedy mohou být také spuštěny na Salomonu. Připravené testbenche nejprve naplní vstupní FIFO frontu veškerými dostupnými daty, které se poté příslušným modulem zpracují. Data z výstupní FIFO fronty jsou posléze uložena do zadaného souboru. Předmětem měření doby výpočtu je však pouze část zpracovávající data bez prvotní a finální obsluhy front v rozhraní.

6.1 Porovnání dekompresorů

Vstupními daty pro všechny typy dekompresorů je opět Silesia korpus sloučený pomocí unixové utility *tar* do jednoho souboru, který je zkomprimovaný pomocí referenčního kompresoru s nastavenou závislostí datových bloků.

Podle grafu 6.1 je referenční dekompresor téměř o polovinu rychlejší než softwarový. Jelikož se v dekompresoru nenachází žádné složité ani zdlouhavé operace, jeho rychlost závisí převážně na práci s pamětí a úrovni režie. Při transformaci softwarové předlohy na hybridní implementaci neproběhly žádné drastické úpravy struktury kódu, ale kompletně se změnil způsob práce s daty – sériově po bajtech. Z hlediska počítačů je manipulace s jednotkami bajtů neefektivní, a proto je rapidní nárůst doby běhu očekávaný. Hardwarová verze, tedy

[1] https://www.mentor.com/products/fv/modelsim/



Obrázek 6.1: Graf dob běhů jednotlivých implementací dekompresoru

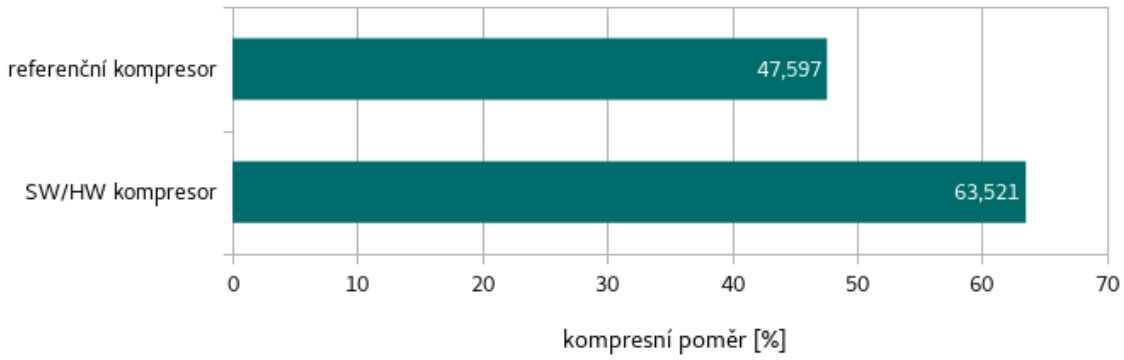
výsledný VHDL kód, je podle simulace až 17x pomalejší než referenční dekompresor. Důvodem tohoto masivního rozdílu je prakticky nulová míra paralelizace zpracování, ze kterého FPGA čipy primárně těží. Velkou roli také hraje pracovní frekvence obvodu 125 MHz, která je 20x menší než u procesoru na Salomonu.

U dekompresoru je teoreticky možné stanovit nejhorší latenci obvodu. Délka doby mezi načtením prvního vstupního bajtu a zapsáním prvního výstupního bajtu do rozhraní závisí na několika faktorech. Archiv může začínat uživatelskými rámci, které je nutné přeskočit, během čehož se žádná data neprodukují. Předpokládejme však, že se v archivu vyskytují pouze LZ4 rámce, neboť se jedná o pravděpodobnější scénář. Na začátku dekomprese je potřeba načíst magickou konstantu a deskriptor rámce. Pak následuje velikost datového bloku, který může být komprimovaný nebo nekomprimovaný. U nekomprimovaného bloku lze ihned po načtení jeho velikosti vypisovat data uvnitř bloku na výstup. V případě komprimovaného bloku se po načtení velikosti bloku musí navíc načíst počet literálů, než je možné literály vypisovat. Nejhorší latence se tedy odvíjí od počtu literálů. Maximální přípustný počet literálů v jedné LZ4 sekvenci, než se vytvoří nekomprimovaný blok, si ale stanovuje každý kompresor individuálně. Záleží tedy na použitém kompresoru. Ze simulace hardwarového dekompresoru lze vyčíst, že zpracování úvodních metadat bloku trvá 15 taktů, další 4 takty pro načtení velikost bloku. Token a dodatečné bajty délky literálů se načítají každý takt. Jakmile je délka literálů známá, za 4 takty je na výstup zapsán první bajt. Za předpokladu, že hranice počtu literálů v sekvenci je větší než 15, lze pomocí rovnice (6.1) vypočítat nejhorší latenci dekompresoru, kterou je možné od archivů daného kompresoru očekávat. Parametr l je velikost hranice počtu literálů a T je perioda hodinového signálu v ns. Pro hranici 1 289 literálů v sekvenci, která je použita ve zde implementovaných kompresorech, a pracovní frekvenci hardwarového dekompresoru 125 MHz je nejhorší očekávaná latence rovna 232 ns.

$$\text{latence} = \left(25 + \left\lfloor \frac{l - 15}{255} \right\rfloor\right) \cdot T \quad [\text{ns}] \tag{6.1}$$

6.2 Porovnání kompresorů

K získání parametrů jednotlivých kompresorů je použit Silesia korpus sloučený do jednoho souboru pomocí programu *tar*. U kompresoru je kromě doby běhu také sledován dosažený kompresní poměr na základě výstupního archivu.



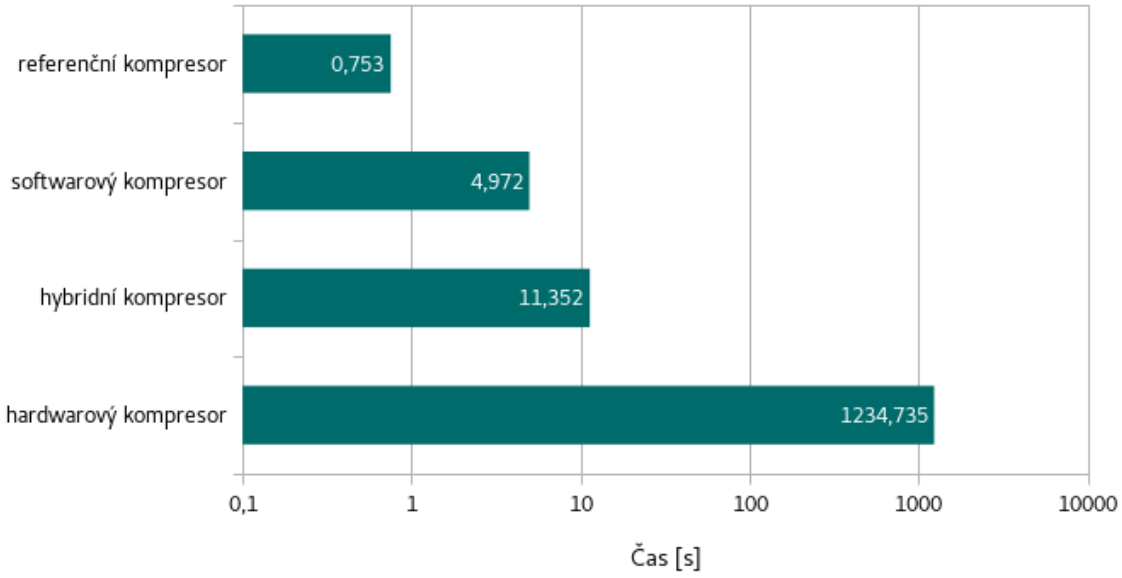
Obrázek 6.2: Graf kompresních poměrů jednotlivých implementací kompresoru

Na grafu 6.2 je možné vidět srovnání dosažených kompresních poměrů referenčního a zde implementovaných kompresorů, které kompresní poměr z důvodu identické vyhledávací metody sdílí. Referenční kompresor nabízí lepší kompresní poměr, což se na testovacím Silesia korpusu projevilo menší velikostí výsledného archivu o 32 MiB. Celkem archiv dosáhl velikosti 96 MiB z původních 202 MiB oproti archivu ze zde implementovaného kompresoru o velikosti 128 MiB. Možné důvody horšího kompresního poměru jsou rozebrány v sekci 6.3.1.

Doby běhů jednotlivých implementací kompresorů jsou vyobrazeny v grafu 6.3 s horizontální osou v logaritmickém měřítku. Optimalizace výpočtu CRC provedená u softwarového kompresoru bohužel k dorovnání času běhu nestačila. K dalšímu zrychlení by byla potřebná podrobnější analýza a pravděpodobně kompletní změna vnitřní struktury kompresoru.

U hybridního a hardwarového kompresoru byla měřena doba výpočtu pouze kompresní jednotky. Jelikož může výstupní jednotka kompresoru pracovat na vyšší frekvenci než jednotka kompresní, nemůže žádným způsobem samotnou kompresi zpomalovat. Závislost jednotek je pouze ve směru výstupní jednotky ke kompresní. Navíc se značně zjednodušilo měření hybridní verze, neboť testbench nejdříve naplní daty vstupní fronty v rozhraní, kompresní jednotka poté všechna data zpracuje a naplní komunikační fronty `DataFifo` a `MetaFifo` z obrázku 5.2, ze kterých následně výstupní jednotka zpracovaná data načítá a na výstupní rozhraní posílá výsledný archiv. V hardwaru by tyto akce probíhaly paralelně, jenže testbench je sériový program, takže pokud by měření zahrnovalo i výstupní jednotku, výsledný čas by nereflektoval paralelismus jednotek, který se v hardwarovém provedení reálně nachází.

Nárůst doby běhu hybridní implementace kompresoru oproti softwarové je očekávaný. Stejný jev nastal i u dekompresoru z důvodu změny stylu práce s daty na ryze sériový, který pro klasické počítačové systémy není efektivní. U hardwarové verze je však pozorovatelné



Obrázek 6.3: Graf doby komprese dat jednotlivými implementacemi

významné zpomalení, které je mnohem vetší než dvojnásobné v případě dekompresoru. Získaný čas není naměřený. Jedná se spíše o odhad pravděpodobné doby komprese v FPGA. Simulace hardwarového kompresoru běžela rychlostí 1 sekunda simulačního času za přibližně 3 hodiny reálného času. Podařilo se odsimulovat 30 sekund běhu kompresoru, během kterých kompresní jednotka prostřednictvím front `DataFifo` a `MetaFifo` vyprodukovala pouhých 3,1 MiB výstupních dat archivu z celkových 128 MiB. Celá simulace by tedy stejným tempem trvala přibližně 165 dní. Získaný údaj rychlosti komprese 3,1 MiB za 30 sekund byl použitý k dopočítání očekávané doby komprese, má-li být výsledný archiv o velikosti 128 MiB. Výpočet kompresoru se tak pohybuje kolem 20 minut. Důvodem je nízká pracovní frekvence čipu 90 MHz, která je o 35 MHz nižší než u hardwarového dekompresoru. Úroveň paralelismu je i v tomto případě v podstatě nulová, což má za následek nevyužití značné části potenciálu FPGA čipů.

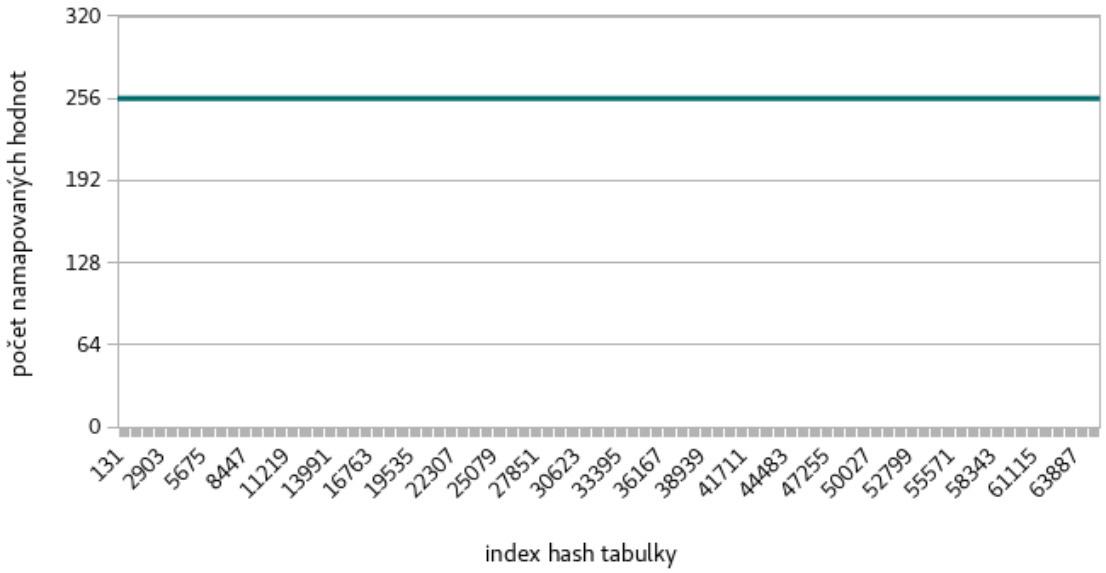
U kompresoru nelze očekávanou latenci jednoduše stanovit. Záleží totiž čistě na vstupních datech. První data se na výstupu objeví po ukončení prvního datového bloku, což může nastat při naplnění celého bloku, při překročení hranice maximálního počtu literálů v LZ4 sekvenci nebo pokud je nekomprimovaný blok předčasně ukončen kvůli nalezení match posloupnosti. Překročení hranice počtu literálů ale není relevantní případ, neboť je blok ze začátku prázdný. Doba vytvoření bloku se velmi liší podle úspěšnosti hledání ve slovníku, takže latenci hardwarového kompresoru nemá smysl uvažovat.

6.3 Návrh optimalizací

Na základě zjištěných parametrů v této prací implementovaných kompresorů a dekompresorů je možné navrhnout dodatečné optimalizace a změny. Zásadní skutečností však zůstává, že algoritmus LZ4 je převážně sekvenční, a tudíž by jeho paralelizace byla velmi obtížná.

6.3.1 Kompresor

Zvolená vyhledávací metoda s hash tabulkou a CRC hash funkcí dosahuje na testovacím Silesia korpusu o 16 % horší kompresní poměr než referenční kompresor s hash tabulkou. Příčinou tohoto rozdílu mohou být jednotlivé komponenty hash tabulky. Jedním z hlavních problémů hash funkcí je distribuce hodnot. Jelikož hash funkce mapuje vstupní prostor hodnot na výstupní prostor, který je obvykle menší, dochází ke kolizím. Pokud má hash funkce špatné distribuční vlastnosti, na některé výstupní hodnoty je namapováno mnohem více vstupních hodnot než na ostatní. V extrémním případě mohou být veškeré vstupní hodnoty namapovány na jednu položku v hash tabulce.



Obrázek 6.4: Graf distribuce hodnot funkce CRC-CCITT (Kermit)

Graf 6.4 zobrazuje distribuci hodnot zvolené CRC funkce pro všechny kombinace 3bajtových vstupů. Je zřejmé, že CRC funkce rovnoměrně rozkládá vstupní prostor hodnot na košíky stejné velikosti 256 hodnot, a problémem nerovnoměrné distribuce hodnot tedy netrpí. Důvodem nárůstu kompresního poměru se tedy zdá být hash tabulka samotná, resp. délka kolizního seznamu v buňkách. Díky snížení maximální velikosti bloku se uvolnilo velké množství paměti, takže kolizní seznamy lze rozšířit a přidat další kolizní bity k indexům tabulky.

6.3.2 Formát datového bloku

Formát datového bloku v LZ4 rámci, tak jak je definovaný ve specifikaci a popsaný v sekci 3.3, není vhodný pro streamové zpracování. Hlavní problém činí velikost bloku, kvůli které není možné zpracovaná data poslat na výstup, a kompresor je tudíž musí uchovávat, dokud není blok ukončen. Zrušením velikosti bloku před samotným blokem by se počáteční prodleva odstranila.

V LZ4 rámci by se nacházel jeden kontinuální blok bez předem stanovené délky, který by obsahoval dva typy položek – drobně upravené LZ4 sekvence a nekomprimované části. LZ4 sekvence by se nijak zásadně neměnila, akorát by dostal neplatný match offset 0 nový význam. Pokud by byl počet literálů a match offset roven 0, ihned za hodnotou offsetu by se

nacházel blok nekomprimovaných dat, jehož délka by byla předem stanovena nebo získána z deskriptoru rámce. Aby bylo možné zachovat alespoň kontrolní součet rámce, bylo by nutné vymezit konec upraveného datového bloku, což by určovala kombinace nenulového počtu literálů a nulového match offsetu v sekvenci. Mohla by nastat situace, že by poslední vstupní data měla být zapsána jako nekomprimovaná, a tím pádem by nebylo možné blok řádně ukončit kvůli chybějícím literálům. V takovém případě by se povolovalo data ponechat jako literály.

Kapitola 7

Závěr

Implementace LZ4 kompresoru a dekompresoru v nástroji Catapult prošla úspěšně všemi kroky – od počáteční softwarové implementace přes hybridní variantu a její funkční verifikaci až k simulaci výsledného VHDL kódu. High-level syntéza je velmi silný nástroj, který umožňuje abstrahovat práci na úroveň softwaru a běžných programovacích jazyků, ale dokáže vygenerovat komplexní hardwarový obvod ve VHDL pro konkrétní FPGA platformu, což urychluje celý proces vývoje. Verifikace je možné provádět již na softwarové úrovni a podklady pro simulace jsou generovány automaticky. Avšak kvůli automatickým transformacím není možné libovolně zasahovat do procesu a uživatel tak ztrácí plnou kontrolu nad projektem. U kompresoru konkrétně nebylo možné využít dostupné LUT na FPGA čipu pro mapování proměnných programu. High-level syntéza zatím nedokáže transformovat libovolný zdrojový kód v jazyce C/C++ na VHDL kód. Je nutné dodržovat jistá omezení v používání datových typů a konstrukcí v programu. Právě proto vznikla hybridní implementace obou komponent z původní softwarové.

Kompresní algoritmus LZ4 je ve své podstatě sekvenční, tudíž žádné významné paralelní zpracovávání neumožňuje. Jelikož jednou z hlavních silných stránek FPGA čipů je právě schopnost paralelního zpracovávání dat, nelze čip plně využít, a proto není tento algoritmus pro implementaci v FPGA vhodný. Jak je možné vidět u srovnání implementací v sekci 6.2, hardwarový kompresor by nenašel žádné reálné uplatnění kvůli mnohonásobné delší době komprese oproti softwarovým verzím. Hardwarový dekompresor by mohl být teoreticky nasazen do prostředí, ve kterém celková doba dekomprese není klíčovou, ale záleží hlavně na latenci.

Literatura

[1] Cards | Liberouter / Cesnet TMC group. [online], [vid. 2017-04-22]. Dostupné z: https://www.liberouter.org/technologies/cards/.

[2] iPXE - open source boot firmware. [online], [vid. 2016-12-28]. Dostupné z: http://ipxe.org/.

[3] Collet, Y.: LZ4 - Extremely fast compression. [online], [vid. 2016-12-28]. Dostupné z: http://lz4.github.io/lz4/.

[4] Cook, G.: Catalogue of parametrised CRC algorithms with 16 bits. [online], [vid. 2017-05-15]. Dostupné z: http://reveng.sourceforge.net/crc-catalogue/16.htm.

[5] Hummel, V.: *Vysílání paketů na 100 Gb/s Ethernetu*. Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2014.
URL http://www.fit.vutbr.cz/study/DP/BP.php?id=16029

[6] International Organization for Standardization: Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model. [online], [vid. 2017-05-15]. Dostupné z: http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html.

[7] Kurt Mehlhorn, P. S.: *Algorithms and data structures: the basic toolbox*. Berlin: Springer, 2008, ISBN 978-3-540-77977-3, 300 s.

[8] Mentor Graphics Corporation: *Algorithmic C (AC) Datatypes, Software Version v3.7*. Červen 2016, dostupné z: https://www.mentor.com/hls-lp/downloads/ac-datatypes.

[9] Pircher, T.: pycrc, a free CRC source code generator. [online], [vid. 2017-05-15]. Dostupné z: https://pycrc.org.

[10] Salomon, D.: *Data Compression: the complete reference*. New York: Springer, třetí vydání, 2004, ISBN 0-387-40697-2, 898 s.

[11] Salomon, D.: *A concise introduction to data compression*. London: Springer, 2008, ISBN 978-1-84800-071-1, 310 s.

[12] Vlček, K.: *Komprese a kódová zabezpečení v multimediálních komunikacích*. Praha: BEN - technická literatura, druhé vydání, 2004, ISBN 80-7300-134-9, 258 s.

[13] Williams, R. N.: A Painless Guide to CRC Error Detection Algorithms. [online], [vid. 2017-05-15]. Dostupné z: http://www.repairfaq.org/filipg/LINK/F_crc_v3.html.

[14] Xilinx: *7 Series FPGAs Data Sheet: Overview*. Březen 2017, dostupné z: https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf.

Příloha A

Obsah přiloženého paměťového média

- compressor – zdrojové kódy LZ4 kompresoru
  - software_version – zdrojové kódy softwarové verze kompresoru
  - hybrid_version – zdrojové kódy hybridní verze kompresoru a testbenche
- decompressor – zdrojové kódy LZ4 dekompresoru
  - software_version – zdrojové kódy softwarové verze dekompresoru
  - hybrid_version – zdrojové kódy hybridní verze dekompresoru a testbenche
- thesis – soubory textu diplomové práce
  - src – zdrojové LaTeX soubory práce
  - thesis_e.pdf – text práce ve formátu pdf (elektronická verze)
  - thesis_p.pdf – text práce ve formátu pdf (verze pro tisk)